UNLIMITED

(2)

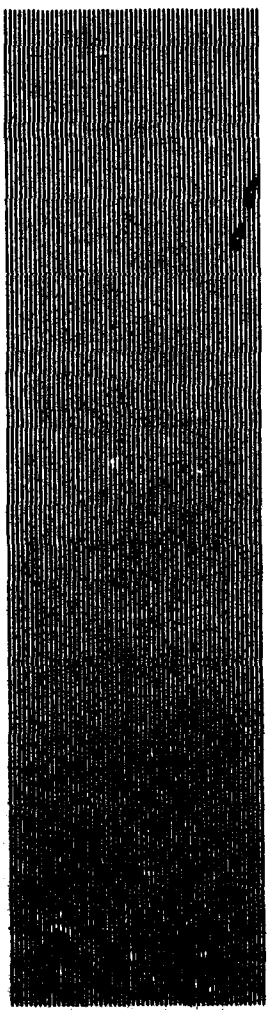
Report No. 89017

Report No. 89017

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

THE RATIONALE AND SPECIFICATION OF THE ELLA GRAPHICS SYSTEM

Authors: C S Wood, N E Peeling, J I Thompson



PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
[illegible]
[illegible], Worcestershire.

[illegible] 1989

CONDITIONS OF RELEASE

0061318 BR-112677

********** U

********** Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 89017

TITLE: RATIONALE AND SPECIFICATION OF THE ELLA GRAPHICS SYSTEM
Version 1

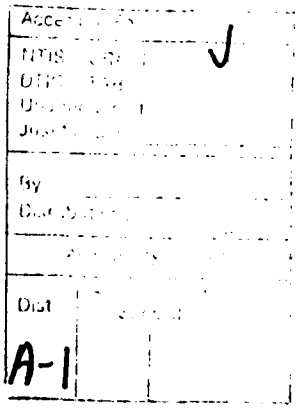
AUTHORS: C.S.Wood, N.E.Peeling and J.I.Thompson

DATE: October 1989

SUMMARY

The ELLA Graphical Language includes graphical representations of most of the high level abstractions in text ELLA, in particular high level data-types and behavioural constructs. The Graphical Editor allows a designer to draw a circuit using these abstractions, which is then compiled into the ELLA system.

Graphical and text declarations can be used in exactly equivalent ways in the extended ELLA system specified here. The Graphical input system thus gives the designer the freedom to use ELLA graphics or text where each is most appropriate, without losing the high level design capabilities when he prefers to use graphics.



CONTENTS

6.3.2.1) Adding Index Text
6.3.2.2) Deleting Index Text
6.3.2.3) Contracting Index Text
6.3.2.4) Expanding Index Text
6.3.3) CASE Test Text
6.3.3.1) Adding CASE Test Text
6.3.3.2) Deleting CASE Test Text
6.3.4) Text Box text
6.3.4.1) ID Box text
6.3.4.2) Other Text Box texts
6.3.5) Pin-indexes
6.3.5.1) Adding Pin-index Text
6.3.5.2) Deleting Pin-index Text
6.3.5.3) Contracting Pin-index Text
6.3.5.4) Expanding Pin-index Text
6.3.6) Show text

6.4) The stack

6.5) Delete Object
6.5.1) Deleting individual objects
6.5.2) Deleting groups of objects

6.6) Duplicate Object
6.6.1) Duplicating individual objects
6.6.2) Duplicating groups of objects
6.6.3) Duplicate object or area n times

6.7) Insert Object
6.7.1) Inserting individual objects
6.7.2) Inserting groups of objects

6.8) Move Object

6.9) Change Size of Object

6.10) Other Operations on objects and attributes of objects
6.10.1) Show Specification
6.10.2) Show formal parameters
6.10.3) Show CASE
6.10.4) Split pin
6.10.5) Place minor pins
6.10.6) Formal to name
6.10.7) Reroute Wireseg
6.10.8) Show direction
6.10.9) Show import context
6.10.10) Change input order

6.11) Operations on Pictures
6.11.1) Search

INTRODUCTION

This document is in 7 Sections. Sections 1 to 3 are the Rationale for the ELLA Graphics System and sections 4 to 7 are its Specification. Sections 4 to 6 cover the appearance of, and operations on, objects defined in the graphical ELLA Language to be implemented in Version 1 of the System. Section 7 is, however, preliminary Specifications for features which should be implemented in Version 2 of the System and are not tested in the RSRE prototype. Experience has taught us that a specification is more likely to be accepted if the implementing team know why the system is as it is. The justifications given in the first three sections are intended to clarify any questionable decisions in the Specification itself.

1) OVERVIEW OF INTENTIONS

The intention of the Ella Graphics project at RSRE was to produce a graphical extension of the ELLA syntax which would be a natural part of a total ELLA design system. We believe it will be used as an alternative input medium from text at all levels of the design process, particularly where there is random logic or where the structure at that level is not describable algorithmically or regularly. In such situations the clear graphical representation of connectivity may be the best way to understand the circuit and thus the natural way to describe it.

The prototype system written at RSRE implements essentially the explicit "MAKE and JOIN" constructs in ELLA, which have to be slick in any graphics package, together with the extra pictures necessary to represent and manipulate user-defined abstract data-types and some of the value-delivering constructs available in text ELLA.

The requirements of the ELLA Graphics project are thus:

1) To specify a Graphical syntax for the ELLA Language.
2) To specify a Graphical Editor for ELLA which highlights the advantages of graphics over text.
3) To specify and implement a Graphical Editor with which to input Graphical ELLA designs into the ELLA System, with acceptable performance and functionality to be the prototype of a commercial system.

We identified various areas containing problems we had to solve and other areas where we decided it was best to avoid the problems rather than solve them - these are discussed in Sections 2 and 3. However, there are three issues which are fundamental to our design approach which need to be stated immediately.

First, as with the whole of the rest of the ELLA project, we have adopted our usual philosophy of designing and implementing at the same time. As a result, the ELLA system, and particularly the Language, is an example of the power of simplicity over the confusion

of complexity. We are hoping to attain the same "feel" with the graphics system.

Secondly, the most important advantage of a diagram over text is its ability to show the connectivity of a circuit at a glance. This implies that the wires are more important than the boxes in terms of understanding the data-flow through the new circuit, so pin positions must not be fixed. Each instance of the same FN need not now have an identical icon. The advantages gained from better understanding of the data-flow may far outweigh any problems of non-identical icons (which, given good naming and highlighting procedures, may never arise) and, of course, the user now has the choice of which approach to adopt.

Thirdly, we have always worked with the data structure describing the circuit being designed. The marks on the screen are simply the visual representation of that data structure at that time. Consequently our editing procedures all work in terms of finding the object under the cursor in the data structure, editing the data structure then drawing the representation of the new data on the screen. The prototype never edits the marks on the screen.

2) PROBLEMS TO BE SOLVED

2.1) Speed, Portability and Density of Information

When we considered the extra information introduced by user-defined data-types, it became clear that the number of distinct pieces of ELLA information on the screen at any time could be extremely large. It was then obvious that this density of information would have consequences for the speed and portability of the software.

The basic problems we had to face for our visually dense pictures were: "given a rectangle of any size, up to and including the whole picture, which objects in the data-structure have a part of their visual representation in that rectangle" and "how do we display and redisplay these objects so that any operation on an object, a group of objects or the whole picture is as fast as possible".

When the graphics project started in 1985 the slowest part of any application was writing to the screen. The advances in graphics support on workstations since then has been phenomenal and now it is less obvious that we would use FLEX on Perq to host the prototype - though its graphics bit operators are still very fast. In 1985, however, the Perq's Graphical operators were extremely impressive, the FLEX environment definitely gave high productivity and, in complete contrast to most (all?) other systems, there was only a single set of coordinates to consider. Regardless of the actual screen position of the picture, the programmer could write everything with respect to (0,0) at the top left of his picture area and any transformations made necessary by the window position or the position of the picture on the screen were dealt with by FLEX. In addition, ELLA was not available on workstations, bit-mapped screens were much rarer, the wealth of windowing and graphics packages available today had not yet appeared and it was unclear that the project would lead to a prototype and specification instead of many other possible conclusions.

The problem of searching the data-structure led us to develop an algorithm based on a simple square "mesh" covering the entire picture which acts as a 2D look-up table into the data-structure. Each mesh square contains lists of pointers to objects which have all or part of their representation in that square. These lists are, of course, much shorter than the lists of all the objects in the picture and therefore much faster to search.

We use the mesh exclusively to find the object under the cursor before performing an editing operation on that object. Because of the efficiency of our algorithm in finding objects, the extra information we need to display to make the pictures represent ELLA has not affected the speed of the system as seen by a user - in fact the speed of an editing operation on an object is independent of the total amount of information in the data-structure and is only dependent on

the density of information in the mesh square(s) the object is in. (This should be compared with the sparse diagrams produced by conventional schematic editors whose searching algorithms operate on the whole data-structure and are still quite complex despite optimising out any empty space between the objects.)

The mesh is also used to solve the second problem - which objects need to be drawn in the clear area of the screen which appears during scrolling, or redrawn after a delete operation. There are situations, however, where deciding what to display via the mesh lists is slower than via the full lists depending on the number of mesh squares involved and the density of information in each. We have carried out some experiments on algorithms for drawing partial pictures and now use a rule-of-thumb that if less than one third of the picture is being drawn we use the mesh lists, otherwise we use the full lists. This gives a respectable drawing speed for all fractions of pictures and all local information densities. It is, of course, implementation dependent so no further work will be done on the prototype in this area.

The level at which the drawing procedures are written is another consideration affecting speed and portability, because it is essential that most operations appear instantaneous. We could have written the prototype drawing procedures at a much higher, machine-independent, level than we did. However, given our equipment in 1985 we chose to use the low level operators and real hardware supplied with the Perq specifically for graphics to write to the screen. At that time we felt that any port or reimplementation should similarly make use of all available hardware support for graphics. Now that writing to the screen is so fast on many workstations, it is likely that the bottleneck has moved elsewhere and it may no longer be necessary to write low-level graphics procedures to gain acceptible performance. This is another implementation dependency - the commercial platforms for the system should use whatever means are available to produce fast enough (ie instantaneous) displaying. Portability, therefore, was not a reasonable expectation for the drawing procedures on FLEX and should not necessarily be for any other implementation.

Another way of speeding up drawing and also increasing the possible information density is to limit the shapes to be drawn to horizontal or vertical lines and bit patterns (1D or 2D arrays) which are trimmed by the visible area of the picture or by the area we want to redraw, whichever is smaller. In the prototype the trimmed arrays are then written directly to the screen, and areas of the picture moved en bloc by the dedicated hardware (eg in scrolling behind a window). We also only allow three line styles, solid, dashed or white (for deletion - there are no invisible objects). The clear horizontal or vertical area which appears on scrolling, or any area to be redrawn after, say, deleting, is then filled in by the low-level

graphics procedures, aided by the speed of the mesh algorithm finding what to draw and the simple shapes making trimming easy.

Scaling is another area where the information density in our pictures could cause performance problems. To avoid this we are only considering one integer scaling factor greater than 1 and one less than 1. This removes the problem of font scaling for text in the picture and allows us to use a maximum of three set fonts.

The standard scale uses a small font and a medium font for its text depending on the object with which the text is associated. The larger magnification factor should allow editing at that scale, as at the standard scale. This size picture uses the medium font and the large font, replacing the small and medium fonts respectively. Since the font sizes are set, the visibilty of names at the larger scale will depend on the magnification factor. An option to truncate hidden names automatically, if the large scale still isn't big enough, should be given so that we have a scale where all names are visible.

The factor less than one should be such that the grid of reachable points covering the picture and the bit patterns for pins etc scale easily - ie a factor of the original grid size. The smallest possible factor is one which gives a grid size of 4 pixels, provided that the picture is still resolvable on the screen at this scale. Editing the reduced picture is not allowed and it should be regarded as an overview of the standard or large size picture, with names, text and pins being represented as dots so that no information ever actually disappears from view. It should, however, be possible to indicate the cursor position, the visible area of the standard or large picture and the results of some global operations, such as highlight, on the overview.

These restrictions on scaling still allow many useful features. The maximum and minimum sizes are resource and resolution dependent respectively. Once the factors are set, the bit patterns which need scaling can be calculated once, thus reducing the work needed to change the scale of the picture at the expense of a small amount of store.

2.2) Integration with the existing ELLA system

It is vital that we produce a good link between graphics and text so that the designer sees a single integrated ELLA system, part of whose syntax is graphical. Fundamental to this appearance of unity is the analogous treatment of graphics and text. There are several problems to be solved here which fall into two categories - those dealt with in the graphics editor and those which interact with the EASE.

First, in the graphics editor, text (names, labels, indexes, pin-indexes, case tests and ELLA text) must be used wherever appropriate in a picture. Also a text declaration which is used as an instance in a picture must be accessible to the user who wants to see the body of that declaration.

Secondly, in the EASE, the user must be able to see the declaration of a FN or MACro whether it is textual or graphical. He must then be able to edit that declaration, recompile (and de-bug) it or drive the simulator from it in the medium in which it was defined. The problems arising from these aims are discussed separately below.

2.2.1) Names, Labels, Indexes and Pin-indexes

Visible text in the picture itself will be names, indexes, pin-indexes and labels. Names will be attached to function instances, brackets, pins and the outer function. In addition there will be the ability to display the FN_name and specification of an instance as it appears in the library. Names which are required in text ELLA will be compulsory in this implementation and any other names will be treated as graphical labels.

We intend to allow two fonts for text at each scale of the picture. The larger font will be used for FN names and instance names and the smaller font for pin names, pin-indexes and index text. At any time when the text of a name or index or pin-index is "too big", for whatever reason, it can disappear explicitly or implicitly behind a symbol which can be interrogated to display the hidden text. (Case tests are always "too big" to be displayed.) The symbols have been chosen to denote the type of hidden text. The spacing of the objects in the picture and the sizes of the fonts available should allow all names, or symbols, to be clearly visible. This approach avoids a large amount of software for scaling fonts - see Section 3.4.

There is an area where we think some help is needed in generating the compulsory names, ie after duplicating large blocks of the diagram. We have provided a way of generating names with (almost) no restrictions on the naming strategy as a whole - see Sections 3.5, 5.9.6 and 6.6.

Pin-indexes will not be compulsory unless there is some ambiguity without them. For the separate fields of a structure there will also be the usual convention "pin-index increases from top to bottom, or left to right" so the user can choose whether to re-route his wires or add pin-indexes. Facilities are provided to allow the user to use pin-indexes wherever he finds them helpful or to determine the TYPE of a wire or pin if it is not indexed explicitly.

Above all in the naming software, we aim to produce a set of

procedures for naming, labeling or indexing a wide variety of objects without springing any inconsistent "surprises" on the user.

2.2.2) Integration with ELLA text

2.2.2.1) Text from the library as part of the hierarchy

It is now possible to regenerate the original ELLA text of a declaration from the intermediate form stored in the library. (D.J. Snell.) This means that textual and graphical declarations can be called up from the EASE for editing, setting simulator monitoring points or other purposes in analogous ways. In particular, from a picture, an instance can be selected and its declaration displayed regardless of whether the declaration is text or graphics. For a textual declaration, a program will be called to regenerate the original text which will be displayed in a window, then edited or whatever in the usual way, and for a graphical declaration the graphical editor can be called on a new display of the original picture - both displays appearing in a window in front of the present picture. This method of looking down the hierarchy will be completely general and consistent at every level.

2.2.2.2) ELLA text in the picture

We provide a "box" with an output in which value delivering ELLA text can be written, via a window, using names already in the diagram as inputs. This is covered in more detail in Section 2.6.

2.2.2.3) Text from a graphical declaration

The Intermediate Language representation of a graphical declaration can be used to generate the equivalent text declaration of the circuit in a standard text file. The graphical information, stored in the Attributes System, is not used. This text can be examined or used in the usual ways - including editing and recompiling into the ELLA system. A scheme for maintaining a connection with the original graphical declaration if it is overwritten by a text declaration is given in Section 4.2.8.

2.2.3) Instances

In any incomplete picture, as in text, the designer is working with the declarations of an entire context at his disposal. The user should be able to work in a graphics window generated from the operating system (like a text file) or from within ELLAview and must be able to see the declarations available in his working context in either case.

It must always be possible to generate a scrollable list of declarations and their specifications, in the working context and others, from the graphics editor alone. The user can then point to

one and indicate that it is to be included as an instance in his circuit.

In addition, if the graphics window was generated from an ELLAview window the user should be able to point to the standard representation of a declaration in an open context window and "carry" that to his picture. This will not show the specification but the user can always use the scrollable graphics menu method (above) if he needs to.

However the new instance was originally defined and now brought to the picture, the user will have to give it a size by specifying its top-left and bottom-right coordinates, then he can place the pins from a "menu" which won't allow him to forget any. As mentioned in Section 1, understanding of connectivity is more important than fixed icons so any first guess at pin positions is likely to be inappropriate in the new circuit. The designer can then use the move object procedure to move the pins before or after wiring them up to obtain optimum understandability of his design.

2.2.4) IMPORTs

Imports are easy to deal with graphically in three ways, representing one or both of the steps needed to import a function textually and then to make an instance of it. Special symbols are used within the FN box to denote an import.

One method will allow the designer to draw an import box anywhere on his circuit. The wires joined to the box and/or the make_pin and specify_type operations to fix pin TYPEs can be used to create pins and define the specification of the function. The designer will be asked to give the FN_NAME as well as an instance name. When the circuit is complete and the graphical data-structure is "compiled" into the library, the empty import function will be spotted and compiled first with "IMPORT" as its body as in text, then it can be used in the rest of the circuit, and any subsequent circuits in the same context, in the usual way. This is equivalent to, for example, "FN NOT = (bool: ip) -> bool: IMPORT." outside the declaration and "MAKE NOT: not." inside it. The user must also compile into the same context "IMPORTS gates: NOT." (RENAMED may also be used.)

Another method is via a menu of declarations in the working context which also includes the option to change context and pick a FN from there. Once the menu has been closed, the chosen FN is drawn in the diagram in the usual way as its specification is known. Or, where the graphics window is inside an ELLAview window, the user can pick up declarations in any context by pointing at the relevant ELLAview icon. He does not then see the specification of the FN but the system can pick it up and continue drawing the instance as usual. These methods

are equivalent to both IMPORT statements outside the declaration and a MAKE statement inside it, and require no further text elsewhere.

2.2.5) Setting Monitoring Points

The regenerated text also allows the designer to point at a word on the screen such that the system knows what the cursor is on. This means that setting monitoring points in the simulator can be done by pointing at text or graphics, and paths can be defined by combining pointing at an object and opening it to display the next level down in the hierarchy. With compulsory names for the objects in the monitoring path it will be possible to write the path to a file in the form required now for simulator input.

2.2.6) Error Messages

Error messages must be passed back to a picture, if that is how the incorrect circuit was defined. Much of the checking done by the text compiler, at least for our reduced set of constructs, can be done interactively. The best example of this is type-checking, which can, and must, be done when wires are joined. We can also check that all inputs on instances are joined and that names, where given, are unique within their scope. Any errors of this kind can be reported on graphically as they are found within the graphics editor where all the positional information is to hand. We also need to consider assembler error messages (if applicable in this implementation) and messages from the simulator which will need graphical information attaching to them if they are to highlight the problem within the picture and not as a separate textual error display. Compulsory names for objects handled by the simulator (Section 2.2.1) will make this much easier.

2.2.7) Attributes

We intend to use the attribute system to hang graphical information onto the representation stored in the library. A detailed study of the problem has not been done so we cannot yet decide between attaching all the graphical information as a single attribute onto the declaration, attaching the graphical information for each object as an attribute of that object, or some other scheme altogether. Nevertheless, it is quite clear that the picture can be regenerated from its Intermediate Language representation and its attributes, so we can have a two-way flow of information between the EASE and the graphics system where it is natural to do so.

2.2.8) Graphics-to-text and "Compiling" the Picture

Of course, we need a smooth route from graphics to the ELLA Intermediate Language for "compiling" the picture and for any back-ends which may be required by users. For the prototype this has been tackled by an (incomplete) graphics-to-text procedure which

can be called on any picture. It pre-processes the data-structure, looking for faults which could not have been checked interactively and temporary features which should be removed, then produces a list of error messages with pointers to the graphical position of the errors if there are any. The user can then step through the list and correct the picture before attempting to compile it again. When the pre-processor has successfully passed the data-structure it goes into a translating routine which produces a file of ELLA text to send to the text compiler giving us the Intermediate Language representation of the circuit. This ELLA text will always be correct. An attribute generating program is also called to associate graphical information from the data-structure with the compiled text.

Of course, this means that many checks are done twice - first interactively then in the text compiler. However, the graphics team at RSRE are not compiler writers so we don't have the expertise to extract the non-interactive parts from the original compiler. Various enhancements are obviously possible - such as producing a minimal compiler or translating to Intermediate Language or to SID output instead of text or by-passing the compiler altogether and translating straight into Assembler Modes. The text route does have the advantage for us that it is quick to write and easy to check by an experienced ELLA writer. See Section 4.2.3.

2.3) Abstract data-TYPEs

The major visible difference between ELLA graphics and other schematic capture systems is user-defined abstract data-types. These necessitate an extra basic object called a bracket and some extra software to do interactive type-checking.

A bracket splits a TYPE into, or combines one from, component TYPEs. Its graphical representation is a line with pins on either side. This compact representation means that a chain of brackets for complex restructuring can occupy adjacent grid lines. Brackets cover the REFORM and CONC constructs which restructure TYPEs in text ELLA. Indexing can also be done using brackets and pin-indexes, though there is also a value changing object, an index, which is used to obtain a single field, eg [3], or a range, eg [2..4], of a TYPE.

In this implementation the designer has available all TYPEs in a context (except bidirectional FNTYPEs in the prototype, though their representation is specified). Any instance drawn in the picture has pins of known TYPE, as the specification in the library can be copied into the graphical data-structure. Other objects have their TYPEs set by valid joins to the object or by explicitly making a pin and specifying its TYPE. When a wire is drawn between such pins or one

pin and a wire, the TYPEs at each end are checked and the connection cannot be made unless they match and are compatible with TYPEs already on the objects being joined. This means that TYPEs can be deduced in many situations from a known TYPE and a join. Joins between two pins of unknown TYPE are never allowed.

2.4) Editing the Data-Structure

Many of the advantages of editing the data-structure instead of the screen have been covered above, particularly in the section on Speed, Portability and Density of Information (Section 2.1). The data-structure itself is a set of ALGOL68 modes which are put together to form a single mode (PICVAL) which describes the whole picture and is tailored to make editing and displaying as fast as possible. Most of the MODEs in PICVAL include a certain amount of redundant information so that speed is not sacrificed by following a long path to the required value. Needless to say, the redundancy is in pointers only.

The information representing every object, such as FNs, WIREs, BRACKETs etc, is kept in a list of similar objects in PICVAL. In addition the picture is divided into an array of areas called mesh squares, each square being represented by a MODE MESH which is a structure of lists of pointers to each object wholly or partially in that square. Thus there may be a pointer to an object in a number of mesh squares - one for every square that part of the object is in. The whole picture is covered by a VECTOR[]VECTOR[]MESH which is also part of PICVAL. This data-structure gives us a quick and efficient way of finding which object, if any, the cursor is on, without looking for pixels on the screen, and allows the searching algorithm to be independent of the total amount of information in the picture.

The mesh and the low-level drawing procedures mentioned in the section on speed (Section 2.1) give us the ability to write fast displaying procedures which take as their parameters the object to be drawn, essentially a bit-map, and a rectangle in which it is appropriate to draw it. The rectangle may be the window, the picture, a mesh square, the boundaries of the object itself, any intersection of these or any other sensible area. This strategy means that we need never undergo time-consuming searches for individual pixels but can always work in terms of meaningful objects in our data-structure (which also generates easily readable programs).

2.5) Reducing the amount of software

Our aim here is to produce a maintainable system.

Rather than end up with something baroque, we have imposed a few useful restrictions to reduce the incredible number of degrees of freedom facing the graphics implementor. In the context of ELLA diagrams these are all sensible, and probably unnoticeable, restrictions. We have produced a system for ELLA and have provided everything necessary to describe ELLA circuits graphically. These restrictions include font sizes, line styles, icon shapes (rectangles), and line slopes and are all discussed elsewhere.

At the moment, the graphics editor is being developed on FLEX. The working environment on FLEX makes it very easy to avoid duplication of code because of the generality with which FLEX objects can be positioned in a file, moved between files and operated on and tested. In addition, version inconsistencies are made impossible by automatic propagation of any changes to the FLEX unit of code (a Module). This results in a system with many modules, each containing one - or sometimes a few - Algol68 procedures. Reorganisation of the procedures into larger units would probably be required on a conventional architecture where linking is necessary but we cannot stress enough how much FLEX has helped us to produce readable, concise code.

2.6) Value delivering constructs

Many ELLA constructs deliver a value. These include FNs, CASE clauses and SEQuences which have a graphical representation, and some others which don't. In this representation of ELLA, objects are used to transform signals which are passed into and out of them via pins or names in scope. Thus there are FNs, CASEs, brackets and indexes, which transform signals between their input and output pins, and also a general "value delivering box", or text box, which holds some ELLA text and has a single output for the value delivered by that text, the inputs being via names in the rest of the circuit in scope at the level of the text.

The text is typed in a window linked to the box and can be any ELLA text which delivers a value of a TYPE available to the circuit. This implementation supports and specifies SEQ, CASE, BEGIN/END and id (identifier). Note that this gives us a textual way of getting CASEs into the picture if a graphical representation is inappropriate.

3) PROBLEMS TO BE AVOIDED AND JUSTIFIABLE RESTRICTIONS

3.1) Missing Constructs

There will be no graphical representation for the following ELLA constructs: MACros, Replication, and FNSETs. These are fairly large extensions which have been thought about to varying extents and are probably all implementable next time round.

3.2) Local Declarations

In this implementation there will be no local declarations of anything except brackets. These may translate to local REFORM FNs. We can see how to implement local declarations of TYPEs and FNs but simply don't have the time to do it.

3.3) Line slopes and styles

There will be no lines of arbitrary slope. All our constructs to date can be visualised using horizontal and vertical lines, with the added attributes of single or double thickness and black or dashed. Even extending from this first implementation to constructs such as MACros, we are sure we don't need any extra lines, except perhaps a few at 45 degrees. The amount of software needed to deal properly with partial redrawing of arbitrary lines is simply too much. (Dashed lines are bad enough, anyway.)

3.4) Zoom

We won't provide infinitely variable zoom. We shall instead provide two scale factors, one (>1) for enlargements, and one (<1) which allows the whole circuit to be displayed on the screen for a global view. Three set fonts of different sizes will be available with the system. The standard view will use the smaller two fonts and the enlarged view will use the larger two - in each case the smaller font of the pair being for pin names, pin-indexes and index text and the larger one being for other names. The reduced picture, an overview which cannot be edited, will have the smallest font as its larger font and "▪" for all text in the smaller font if the scale factor allows. Of course, the reduction may be such that no text is displayable at all in which extreme case a dot can be used to indicate the presence of text in the appropriate place. A lot of difficult software has thus been avoided without losing any information from the picture.

3.5) Auto-generation of names

Names will be compulsory where they are necessary in text ELLA. Auto-generation of names is another well known problem area which we wish to avoid. By insisting on names for ELLA objects we can specify internal names for all pins, eg an input of instance "fred" is formalOFfred (where "formal" is found from the declaration of the

FN), or its second output (second field of output of declaration) is fred[2]. Where an instance is made by duplication we can add a suffix to its name, eg fred.2, and the user can change this if he wants. This feature will be necessary where large blocks of FNs are being duplicated, to avoid a lot of tedious naming by hand. There are problems here which may just have to be faced - these are explained and the solutions justified in Sections 5.9.6 and 6.6.

3.6) View Planes

Our system won't have view planes. We need to be able to pick up the object under the cursor as quickly as possible. This is much easier when there are no view planes and their omission is no great problem in circuit diagrams as components don't overlap. Occasionally we have discrimination problems anyway (temporary wires over FNs or FN, PIN and WIRE having a grid point in common) but these are resolved by a sensible algorithm for searching the data-structure.

3.7) Text-to-graphics

There will be no text-to-graphics procedure. The problem here is essentially one of layout which is an entirely different area from our work. Consequently there will be no auto-routing procedure. Even Mentor see this as a luxury.

3.8) Icons

Icons, either standard gate icons or user defined symbols, are not specified here. This is because they can limit pin placement, thereby obscuring connectivity, and because instances in the new diagram come from a specific ELLA context, not from a general library of components. An icon editor may be added as an extra feature (not under the initial contract), though there must be restrictions on its use. For example, using the most common type of icon - a gate - an AND gate has one straight side on which input pins can be placed and one curved side on which the output pin is placed. All the pins can be on grid points, as we require, and the icon can be rotated through multiples of 90 degrees with no problems. However, instance names, the FN_name and any other text which is allowed inside the instance (timescaling, Import, BIOP or alien code information and pin names) will cause problems which must be addressed before agreement to include the icon editor is given by RSRE.

3.9) Reduced detail overview

It should be possible to provide a reduced detail view of the whole design (rather than reduced size without loss of information) so the user can get an overview of the connectivity or structure of the entire circuit and find his way about it easily. This poses the problem of deciding what information becomes superfluous in the

context of an overview and can therefore be omitted at a lower level of detail. We are not addressing this problem now.

3.10) Freezing the specification

We won't freeze the entire specification of the system too soon, if at all. Good ideas which weren't obvious from the start should go in and bad decisions whose consequences weren't forseen should be reversed wherever possible. Implementation of the prototype has been invaluable in writing the specification and, as long as we continue implementing new features or improving existing ones, we will continue to learn more about the spec.

4) ELLA GRAPHICS SPECIFICATION - PART 1

The ELLA Graphics System is a schematic capture package for the ELLA language as it stands at Version 4. It exists in a window which can be called from the host operating system or from ELLAview. It has its own menus, error messages and displays, and a graphical syntax for describing ELLA designs diagramatically. It is very closely linked to the rest of the ELLA system and is not intended to be a general schematic capture package.

This section specifies the user-interface, how the Graphics System interacts with the rest of ELLA and the performance of any new Graphics System implemented from this specification.

At RSRE we have a prototype of the graphics system though it is not linked to the rest of ELLA. Members of the team implementing the specification are expected to be familiar with this system as an aid to understanding the spec, and may like to port it to one of their own workstations to have it, and the source code, available at their workplace. Apart from obvious machine dependencies, eg windows, menus, scrolling, writing to the screen, the code should port easily via the ALGOL -> C translator.

4.1) The User-interface

The entire Graphics System occupies a rectangular window on the screen. In that window there is an inner window showing a view of a rectangle in which the circuit may be drawn. This inner rectangle may be any size at all. In other words it must be scrollable so that a picture area bigger than the internal window can be defined. The minimum size of the inner rectangle is 6x5 grid points - the grid being defined in Section 5. The rest of the Graphics window is occupied by a set of rectangles containing the command words for the operations specified in Section 6 and acts as a static menu. These command rectangles should form the top and right border of the inner rectangle.

4.1.1) Initialisation

The Graphics System can be initialised from the host operating system or from ELLAview. The parameters it needs are the size of the picture area - a coordinate pair in multiples of the grid spacing - and an ELLA context name which becomes the working context. From the operating system the entire ELLA library must also, presumably, be specified before the context name is meaningful. The size of the outer graphics window is the first thing to be set on the screen by the user. The static menu is then scaled to fit the dimensions of the outer window (a minimum size will need to be set when the menu layout has been finalised) and as much as possible of the picture area is drawn in the inner window. An empty picture is specified in Section

5. The top left of the picture area (0,0) should be at the top left of the inner window when the system is initialised - see Section 4.1.2. Other corners of the picture area are only visible without scrolling if the inner window is bigger than the picture area in one or both dimensions.

4.1.2) Coordinates

The user should only be aware of a single set of coordinates with respect to the top left of the circuit area. This point, (0,0), need not be visible as the user can scroll the picture around behind the inner window to bring any part of it into view. All manipulation of the coordinates of visible areas of the picture, window coordinates, screen coordinates or any other set should be hidden from the user.

I have specified the top left of an area as being its origin. If the ELLA Graphics System is implemented on a machine where the user always expects, say, the bottom left corner to be the origin then this will be changed with RSRE's agreement. The origin will not be changed to suit any internal conventions used by the implementors and hidden from the user.

4.1.3) Colour

The commercial implementation must work in monochrome. Colour is not precluded but should be a useful enhancer, not a necessity. There are occasions where colour would definitely be nice, eg temporary facilities like highlighting, or for showing the type on a pin or wire by colour coding the types available and having the colour as part of the data-structure of the picture. These issues are open for discussion once a monochrome implementation exists, though there is no doubt that the design of the objects as specified here will not be changed to let colour be a primary attribute.

4.1.4) Input to the system

There is a three button mouse to drive the static menu, and any pop-up/pull-down menus, as well as to point at the screen. The buttons on the mouse are called select, result and quit throughout the specification. Moving the mouse on the pad/tablet moves a pointer on the screen. This pointer is not the cursor.

The cursor is moved to the position of the pointer by pressing the select button. The cursor itself is x-shaped in the inner (picture) rectangle and can only be placed on grid points. The position of the pointer is therefore rounded to give the nearest grid point. In the static menu area, or in any pop-up menus, the cursor is a rectangle covering the whole area of the command rectangle selected. The command word is then shown in reverse video.

The system should remember the position of the cursor in the data-structure so that, when the user logs out and logs in again the picture can be centered about the last cursor position. This saves scrolling to find out where he was working last time. The same should be true, if possible, between edits of text windows, particularly for text boxes which may contain substantial amounts of text. (This may not be possible as a proprietary editor will be used to enter the text and it may then be stored as an operating system file. This implementation may not remember the cursor position, or even allow it to be remembered in the graphical data-structure and the text file scrolled to centre on that position when it is opened.)

The result button is always used to exit from an editing procedure or to terminate a loop therein. Result never throws away any information and is, therefore, not always meaningful. Where more work has to be done to get an operation to a point where result can be pressed, the user must be given a warning message telling him what to do.

All operations called from the static menu repeat until the result key on the mouse is pressed. For example, entering the picture to carry out a make_input command means that the user can place, or try to place, input pins until he presses result. Or, similarly, the delete command means that the user can delete, or try to delete, a succession of objects without leaving the picture to return to the menu until he presses result.

Quit results in a loss of information and is not always possible. In fact, quit is only possible where the system "asks" the user what he wants to do next - eg go on or quit. It is not possible, therefore, to quit by mistake if the wrong button on the mouse is pressed. Quit leaves the system in the state before the current operation began. It is more suitable for use in operations like move_object where the same object can be moved repeatedly than in the make_? procedures where many similar objects can be made in succession and quit would loose all of them.

Each mouse button should only be clicked once for each operation. (Where the system is hosted on a machine without a three button mouse RSRE should agree the interface to be used. It is possible that two buttons and the space key on the keyboard (for quit) could be used or, on a single button mouse, one click, two clicks and the space key)

Apart from entering text, the keyboard is not used.

4.1.5) Adding extra information

There are three situations where the user has to give extra information to the system - entry of text within a procedure, entry of

a parameter to a procedure and selecting an option at a decision point within a procedure.

Text and parameters are entered in a "text window" one character deep and about 20 wide (but scrollable sideways for longer strings), and with a character-sized cursor. The text window appears next to, but not overlapping, the object requiring text, or next to the command rectangle of the operation which requires a parameter. Text must be valid ELLA text in the relevant context and parameters must be in the correct form for the procedure being called from the current command rectangle. (For example "(Int,Int)", (eg (30,50)), for change_size which changes the size of the picture area, or "Vec Vec Char", (eg ["AND","OR","NOT"]) for search which needs a vector of recognisable strings.)

When the text or parameters have been typed the user presses result to close the window and the entry is checked for validity. If it is invalid, the user can select the same object or command rectangle again in which case the blank window reappears. A valid parameter makes the procedure run. Valid text is added to the data-structure and is drawn. Further attempts to add text to the same object result in a window appearing containing the existing text which can then be edited.

The third situation in which extra information is required is within an editing procedure where a menu may have to appear giving the user choices of action from a decision point. These menus appear in the picture area, not overlapping the object being operated on. For example, if the user is in the naming procedure and has selected the major pin position of a bracket, a two or three line menu will appear adjacent to the bracket giving him the options "Name major pin"/"Name Bracket" or "Name major pin"/"Name Bracket"/Name minor pin" together with a message on the warning line asking him to press result with the cursor on the required line in the menu. Once the object to be named has been chosen a text window appears and the text is entered/edited as above.

The system has a stack which holds some types of object and groups of all objects on a last-in-first-out basis. Procedures which operate on the stack are not properly specified because we don't know yet what people will want. If there are a lot of them it is possible that they will be accessed via a pop-up/pull-down menu from the command "STACK" in the static menu, rather than one command rectangle for each procedure. Some may need parameters which will be entered by a text window as above. (See Section 6.4.)

4.1.6) Carrying out an editing operation

To carry out an editing operation the user moves the pointer to the rectangle in the static menu containing the required command word

and clicks the select button. The cursor moves to the command rectangle which now appears in reverse video. A prompt appears on the warning line telling the user what to do in the picture. The user then moves the pointer into the picture area and clicks the select button to move the cursor (now a "x") to the nearest grid point to the pointer and this becomes the first point required in the new editing operation. The selected command rectangle remains in reverse video as a display of what the user is doing until the procedure is finished.

The command area cannot be reached by the cursor until the editing operation is finished, signified by pressing result (or quit, if this is an option) on the mouse. Any attempt to move the cursor outside the legal picture area for that operation results in the "x" being placed on the nearest legal grid point and a warning message. If the operation is indeed finished (result, or quit, has been accepted) the reverse video in the command area is switched off and the cursor remains where it was in the circuit area until the user explicitly moves it elsewhere. Other attempts to end an editing procedure prematurely generate an error message telling the user why he has not finished the operation and allows him to do so.

4.1.7) Creating Objects

ELLA Objects which can be made, edited and manipulated by the editing operations are: FN Instances, Pins, Wires, Brackets, CASEs, Indexes and Text Boxes. Names, index texts, pin-indexes and CASE tests, which are attributes of ELLA Objects, can also be handled.

There is a distinction between graphical objects and ELLA objects in this spec as follows. ELLA objects are the graphical equivalents of syntactic elements which exist in text ELLA, eg drawing an instance => MAKE, drawing a wire => JOIN, the outer function => spec of declaration and so on. These objects are stored and manipulated by the ELLA Graphics System. Graphical objects are part of the graphical representation of ELLA objects but have no connection with the ELLA language. For example a box (graphical object) is part of the graphical representation of an instance, CASE etc, or a zigzag line is part of the graphical representation of a wire.

Where an object includes a graphical box or a line some indication of its size and shape should be given while the user is defining it. In the prototype we have used small "marker boxes" which are drawn around points chosen as part of an incomplete graphical structure, eg at the first corner of a new box or at each corner of a wire. These disappear either just before the new structure is drawn or when the user gives up for any reason, and are entirely satisfactory in most cases. Continuous drawing of lines or boxes while the select key is depressed, with the final drawing being done once the key is released was considered, though this raised the problem of continuous

deletion and, therefore, redrawing of intersected objects as well. Where we are specifying continuous drawing, because marker boxes are insufficient, is for the temporary boxes used to define an area to be duplicated or deleted. This will make it much easier to see exactly what is in the box so the user can choose whether to go on or quit. (See Sections 6.5.2 and 6.6.2.)

4.1.8) Warnings and prompts

There should be a scrollable (sideways) "line", eg at the top of the screen or graphics window, for displaying warning messages or prompting the user to take some action. This line is not reachable by the user and indeed should be out of the graphics area altogether so that no such message can overlap part of the picture which the user may want to see while he considers the message. Warning messages and prompts are part of the spec and appear after an error or to prompt the user to make a choice. They are visually distinguished by case - warning messages are all in Upper Case and prompts are in Lower Case with capitals where appropriate in English or in ELLA, eg "BOX TOO SMALL" or "Press Select ... or Result ...". Both messages are accompanied by an audible "beep" to draw the user's attention to it. They are written in the font used for instance names in the standard size picture. The messages should be short enough to fit on the line without scrolling it except where their length is unpredictable, eg messages involving TYPEs whose names are user-defined. All messages should be in the style of text ELLA messages, indeed many may be identical to the message in the equivalent situation in the text language. The messages disappear as soon as they are no longer meaningful.

4.1.9) Displays

Displays are to give the user information deemed useful by the designers or asked for by the user himself. Depending on the length of the message, the warning/prompt line can be used or a (scrollable) window can appear showing the complete display which may be quite large. Displays are written like prompts, Section 4.1.8, but the beep is only heard when the display wasn't requested by the user. The cursor remains where it was outside the display line or window - displays are not editable.

The wording of displays given by the system should be short enough to be visible on the warning/prompt line, possibly with some scrolling where user-defined names are involved. For example, where the TYPE checker sets TYPEs on all test pins of a CASE clause because the output TYPE has been set, a message appears of the form "Test Pin TYPEs have been set to ....." where the system fills in the TYPE name. The user should not have to delete explicitly messages which he didn't ask for - this is why they should be on the warning/prompt line rather than in a window which he would have to press result to

close. Instead, these displays are accompanied by a beep and disappear when the user next inputs a signal to the system.

Other displays are asked for by the user and should be on the warning/prompt line if they are only a single line of information, or in a window if there are many lines. The window should be at the top left of the graphics window, overlapping the visible part of the picture area if necessary, and should be scrollable if it needs to be bigger than a maximum size related to the size of the graphics window. (Again, this needs to be used before exact sizes can be specified.) For example, the result of show_TYPE on a pin or wire would be a single line of text, possibly needing scrolling, on the warning/prompt line, whereas the result of show_TYPEs will be many lines of text in a window. There is no beep heard with these displays as the user has asked for them. Those on the warning/prompt line disappear as above for system displays, those in windows are removed when the user presses result.

4.2) Integration with the existing ELLA system

4.2.1) Hardware

The commercial implementation should be available on the machine(s) specified in the contract. This is not a requirement for portability if more than one machine is named - indeed RSRE believes that a completely portable system of this complexity would not meet the performance specification on all machines. Thus a system initially developed for a SUN4 which has very fast graphics capabilities does not need sophisticated algorithms to obtain a satisfactory performance, but the same software may run too slowly on another machine for RSRE to accept it as ELLA Graphics. All versions of this specification running on any machine must meet the performance specification. This is outlined in Section 4.3 which also mentions a formal test set of operations covering all aspects of the system, to be issued later.

4.2.2) Storing a picture

The data-structure holding the picture must be storable in a file under the host operating system, as are ELLA text files before compilation. Although the diagram can be stored on the host machine in an incomplete state, it cannot be passed into the ELLA data-base until it is "compilable". This is exactly analogous to the contents of text files which are not in the ELLA system until they are compiled.

4.2.3) "Compiling" the picture

While the picture is being developed many interactive checks are possible. The most important of these is TYPE checking, which is a vital part of ELLA graphics. The TYPE checker ensures that all TYPEs in the picture are consistent at all times, following the same rules as in the textual language. ELLA objects are checked as much as

possible as they are created and edited, though temporary and incomplete objects are always possible and must be flagged in the data-structure so they can be dealt with later.

Text in the system falls into three categories according to the amount of interactive checking which is done - fully checked, syntactically but not semantically checked and unchecked. Names and index text can be fully checked for case, characters, scope and semantic meaning, eg an index text "[2]" on an input TYPE of (bool,int) can only be possible if the output TYPE is int. Test texts in CASE clauses are checked so that the basic values in the tests are consistent with the TYPEs on the chooser and output (if applicable) of the CASE clause and are also checked against each other for disjointness as in the text compiler.

Once the user has "finished" a design he calls a graphical compiler to check the data-structure and enter it into the ELLA system if it is correct. It is not necessary to close the graphics window, like closing a text file, before compilation - the data-structure should be available to the compiler with the window open. The window will not, therefore, need reopening if there are errors to be corrected.

Graphical compilation happens in three stages though, if the picture is correct the user will not be aware of them. First, a pre-processor is called on the data-structure to find all the constructs which will not translate to correct, compilable, ELLA (eg temporary objects and unjoined inputs) and enter them in a sequence containing the error message and graphical information about the site of the error. This sequence should be user-readable, eg by displaying in a scrollable window not overlapping the picture if possible. This is analogous to editing text files in response to a list of compiler error messages. Procedures should be provided to move the cursor to a given coordinate (including scrolling the picture if the coordinate is not on the screen) and to give the current cursor position so the user can see where he is in relation to the next error site he wishes to visit. (Sections 6.11.10 and 6.11.11.) Once all the errors have been corrected, the user calls the graphical compiler again for another attempt.

Once the pre-processor has been passed, the second procedure is called to transform the data-structure into a form which can be stored in the ELLA system. It effectively compiles the graphical data-structure into ELLA Assembler Modes (and may, in fact, be the text compiler (or a sub-set of it which only does the checks which have not been done interactively) operating on the textual equivalent of the data-structure, or it may be another form of translator direct to Assembler Modes). This is effectively a compilation of correct ELLA. Then a third procedure arranges the graphical information as a set of attributes associated with the declaration or with individual elements within it.

We do not specify a language for the commercial implementation. We only specify that the output of the graphical system is either ELLA Assembler Modes and a set of ELLA attributes holding the graphical information, or something which can be translated automatically into this form, for example a file of ELLA text and associated graphical information to pass to the standard text compiler and to an attribute generator. Error messages must be passed back to the picture as described above, whatever the implementation. There are other levels at which the graphics output could enter the existing ELLA system with agreement from RSRE. Members of the implementing team must be prepared to visit RSRE to become familiar with the structure of the ELLA system and published interfaces to it.

4.2.4) Back ends

Once the picture has been successfully "compiled" into the library the user can ask to run the simulator (or other back ends). It must be possible to regenerate the original picture from ELLA Assembler Modes and attributes so that the user can set monitoring points etc graphically and so the simulator can relate its messages (eg change of signal at a monitored point) back to the picture. Note that the picture needs to be regenerated even if the simulator is called immediately after compilation when the original picture may still be on the screen. This is because the original picture has the same status as a text file whose contents are meaningless to ELLA, whereas regenerating the picture from internal ELLA information means that all the necessary data-structures and values are available for use.

4.2.5) ELLAview

User-readable information about the ELLA library should be obtainable from ELLAview in the same way as when a text file is being prepared. When the Graphics window was generated from the operating system there will be no passage of information between ELLAview and the Graphics System, although an ELLAview window may be open in parallel with the graphics window for examination. However, when the window was generated from within ELLAview, the user must be able to pick up instances from open contexts in ELLAview and, therefore, FN specification information (at least) must be transferable between the two systems.

4.2.6) Hierarchy

The user can point at an instance in his picture and ask to see its body. This will be a text window, if the instance was defined textually, or a graphics window if it was defined graphically. Both sorts of window should allow the user to edit the declaration if he wants, then close the window and recompile the declaration to amend it in the library. He can travel as far down the hierarchy as he wants, pointing at boxes or instance_names at each level.

This facility allows the user to examine and edit individual declarations whenever he wants without leaving the graphics system and also allows him to set monitoring points along a path which can be sent to the simulator.

4.2.7) Graphics to text from the Intermediate Language
A graphical declaration can be transformed into a file of text using the Intermediate Language representation of the circuit and the usual text regenerating program. It needs none of the attribute information giving graphical position, labels etc. This text is, of course, valid ELLA - like any other source text - and can be examined, edited and so on. See Section 6.11.8 - Picture Text.

4.2.8) Overwriting a Graphical Declaration
If a declaration is compiled such that it overwrites a graphical declaration of the same name, the attributes of the graphical declaration and its current Picture Text output may first be dumped to a file together. This file can be used to regenerate an overwritten picture in its original form. This facility should be provided in case the original operating system file holding the picture has been lost by the user - it then provides the same sort of protection as for a text file which can already be regenerated from its Intermediate Language form. See Section 6.11.9.

4.3) Functionality and Performance
The functionality of the commercial system should be no less than that of the prototype. The Specification includes some unimplemented features which are to be included in the commercial system. Any functionality over that specified may only be added by agreement with the Design Authority at RSRE. The Spec makes clear which areas are open for technical debate and subsequent agreement between Praxis and RSRE. Other features may be changed after agreed change control procedures have been followed and RSRE has been convinced.

The performance of the commercial system should be at least equal to that of the prototype. FLEX on Perq is a slow machine, though the Perq has fast operations to write to the screen. It should not be difficult to equal or improve on our performance statistics on more modern hardware with economical code.

The specification of the system stresses that the basic graphical constructs are simple in form but the whole picture may contain very dense information. Our algorithm for searching and displaying the picture, outlined in Section 2.1, is only dependent on the density of information in an area local to the site of the operation and, in the case of the displayer, on the new area of the picture being displayed.

It is not dependent on the total amount of information in the entire picture. Searching and displaying in the commercial implementation should also not be dependent on the size and complexity of the entire data-structure and should show as good a performance curve for large and/or dense circuits as the prototype so that finding objects, displaying and scrolling the picture are acceptably fast.

To formalise these requirements we will provide a set of operations on a given picture which will exercise each operation in a controlled manner and from which we can compare the commercial system with the prototype.

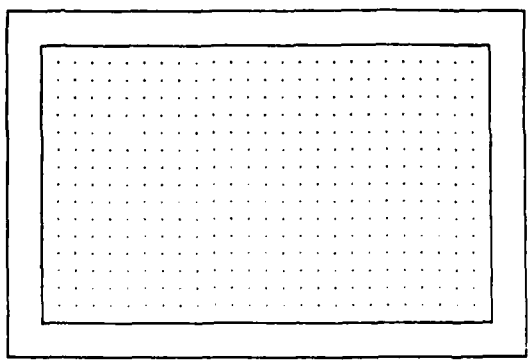
5) ELLA GRAPHICS SPECIFICATION - PART 2, THE PICTURES

INTRODUCTION

The graphics input occurs within a window generated either from the host operating system or from within ELLAview. The window must be big enough for the static graphics menu around its top and right edges (scalable down to a minimum size - see Section 4) The rest of this window is a sub-window onto the rectangular picture area whose size is user-defined when the outer window is created but must be an integral multiple of a grid spacing which is used throughout the system. (The RSRE prototype uses a grid of 10 pixels on a screen with 100 pixels/inch. This spacing is large enough to accommodate the graphical objects clearly and it is easy to hit a grid point with the mouse.) The picture area may be bigger or smaller than the sub-window through which it is viewed, the outer window or the screen on which it is displayed. The user can scroll the picture area in two dimensions behind the sub-window to bring any part of it into view. The top left of the picture area cannot be moved down or to the right past (0,0), nor can its bottom right be moved up or to the left past (xsize,ysize) as there is no need to show blank areas around the picture area. These conditions set the limits on the scrolling software. The static menu is not scrollable.

The grid, the available fonts and the symbols specified for names, pins, boxes etc are all designed to allow the information to be as dense as possible without overlap of any of the objects (except some pin names and pin-indexes which are crossed by wires joining to those pins - the wire going invisibly "under" the text).

The picture area is in two distinct parts. There is a 2 grid wide frame around the area which is used for attributes of the specification (outer function) of the circuit being defined - these are pins, formal parameters (input pin names), other labels on pins and a circuit name. The rest of the area is for defining the body of the FN and is covered in an array of dots at each grid point in x and y. The cursor (a diagonal cross 9x9 square in the prototype) can only be placed on one of these grid points.



5.1) Instances

An Instance is usually a representation in a new circuit of a FN which already exists in the context the user is working in. Graphically it must comprise a FN box, a set of pins, a FN_name (from the library), and an instance name, and may also need some other special symbols - see Section 7. An Instance may also represent an Imported FN from a different context, in which case an extra symbol is present.

The size of the box is user-defined though there is a minimum size for each FN depending on the number of pins. The smallest possible box - a FN with 2 pins and room inside the box for their names - has a perimeter of 8 grid points. Instances of functions must have a perimeter long enough for all the pins to be placed round it. Box sizes are set when the instance is created but can be changed later so that the pins can be moved for optimum wire routes.

Pins appear on grid points on the outside of the box and are solid triangles (their TYPE is always known) - input pins point into the box and output pins point out of it. Pins cannot be placed on the box corners and two pins can never be coincident. The formal parameters can become the pin names of inputs if required, or other names can be chosen. All pins must be placed on the instance so this is enforced when the instance is created. It is, of course possible to move the pins later, before or after connection to another object.

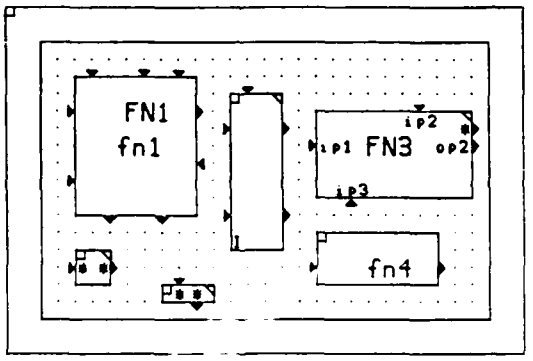
An ELLA FN at the moment has only a single output pin. Graphically it is clearer to split this into its component fields. When pins are being placed on a new instance the user is offered the choice of placing the single output pin or each of its fields individually. Only a single layer of structure is removed - further restructuring can be done with the bracket or index constructions decribed later. See Section 6.10.4. Pin-indexes may then be necessary if there is any ambiguity.

The FN_name is represented by the small hollow square in the top-left corner of the box when the instance is made. It is accessed by putting the cursor on that corner. It can be expanded to visible text if required, and if there is room, when it is written in upper-case letters in the larger font.

The instance_name is user-defined and mandatory. It appears either as lower-case text in the larger font or as a hollow triangle in the top-right corner of the box. The section on names deals with adding the instance_name and general operations on all names.

An instance imported from a context other than the working context (see Section 6.2.3) has a small "I" in its lower left corner if the original context is known, or a small "?" if it is not - the symbol being small enough not to interfere with hidden pin names.

All instances in the prototype implementation have been previously compiled into the library. Local declarations and instances thereof will be specified later (Section 7.3.2) but not implemented at RSRE. A solid box, a solid triangle and bold type will eventually be used for the FN_name and instance_name of a local FN.



5.2) Outer function

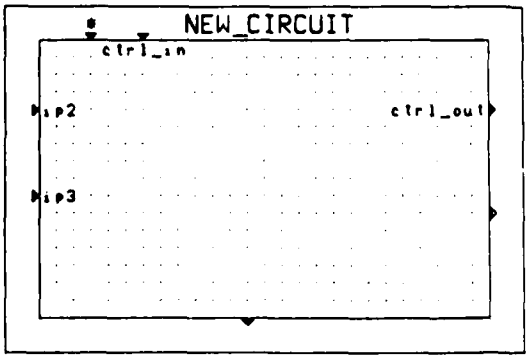
There is a blank 2 grid wide area between the boundary of the diagram and the area in which components can be placed which represents the outer function, or specification, of the new declaration.

Pins are placed along the outside of the inner boundary of this area either explicitly or as the result of a join to the inner boundary where no pin already exists. These pins may or may not be of known TYPE while the function is being declared, so are drawn solid or hollow as appropriate. Eventually, before the picture can be compiled, all the pin TYPEs must have been explicitly or implicitly specified.

Input pins on the outer function must usually be named, as the names are the formal parameters which are usually required in text ELLA. They appear in the spec in the order in which they were created unless this is changed explicitly by a call of a change_ip_order (see Section 6.10.10).

Output pins may be named if required. (These names are just graphical labels at the moment and are stored as attributes with other graphical information. Named outputs may be added to text ELLA and then they will appear in the Intermediate Language representation of the declaration with other ELLA names.) Pin-indexes must be added to the output pins as they have to be combined to an ordered collateral by the translator.

The outer function must be named as the ELLA context requires a unique name for each declaration. The name may be displayed anywhere along the top or bottom border of the picture or contracted to a hollow square in the top left corner of the border area. (A solid box or bold type will be used for the circuit name of a local declaration within a circuit when it is being defined in a sub-window, as well as for its local instance name when the window is closed - see Section 7.)



5.3) Pins

Wires are joined to objects by pins. These objects can be instances, the outer FN, brackets, CASE clauses, indexes, or Text Boxes. There are special rules for adding pins to most of these objects - it is unlikely that these will cause problems because the user will understand the use of each object in the ELLA context which all the component-building rules in the system reflect.

Pins on all objects except brackets and indexes are always represented as solid (specified TYPE) or hollow (unspecified TYPE) triangles pointing into an object, for an input, or out of an object, for an output, of that object. Each pin occupies a rectangle whose dimensions are such that pins on adjacent grid points on the same object do not touch and each pin does not protrude as far as half way across a grid square. This means that an object can be placed on the grid line next to another object with pins and there is no possibility of overlap of the components. In addition a wire between two pins on different FNs but adjacent grid lines is visible. With a grid of 18 pixels we use a pin size of 7x4 - 7 along the box side and 4 orthogonal to it.

Brackets have a visible major pin as above and minor pins represented by a short spike in the direction of the pin which is covered by the wire when the pin is joined. A minor pin of unspecified TYPE has a small (5 pixel) bar drawn across the line.

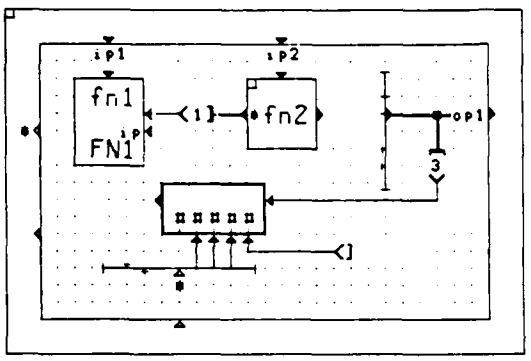
Indexes each have one input, [, and one output, >, with an integer, range of integers or * (hidden text) between them. A compound index, eg [3>[1..2>, still has only one input and one output as joins cannot be made to the component parts of an index. The pins show the direction of flow through the index which can have the obvious 4 orientations on a wire. Index pins of specified TYPE are drawn in bold ([or >). One pin of an index is always joined and this pin is always of specified TYPE.

Text Boxes only have a single output pin, represented in the usual way, as the text will only allow a single "unit" to be delivered. The output can be split as usual so pin-indexes may be necessary. Their inputs come from names in scope outside the box and are not represented as pins.

All pins except test pins on CASEs can, in theory, be named though most of the names will be graphical labels rather than ELLA names. In practice there will be some rare occasions when bracket, index and outer function pins cannot be named - if they are too close to another object. This is because the shape of these objects means that the names must appear outside the object in the general picture area. These cases are always spotted and the user warned. Named outputs of instances from which fanout occurs are passed into the ELLA

system as LET names (though all such outputs need not be named) and names on inputs to the outer FN (formal parameters) are required by the system wherever they would be required by the text compiler.

Some pins can have a pin-index, eg [3], attributed to them to show their place in a structure. This is used where an output of an instance is split up, for multiple outputs on the outer function or text box, or where a bracket combines or splits signals such that there may be some ambiguity about each component. These are illustrated with their graphical specification in Section 5.10.



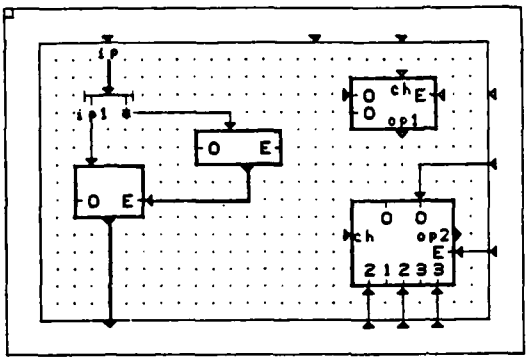
5.4) CASEs

A CASE clause is made by defining a box drawn with double thickness lines. There is a minimum size for a CASE box which is checked when the box is defined, to allow for the output and chooser pins and at least two test pins whose tests each occupy a grid point within the box. The output and chooser (input) are ordinary pins which are made and operated on in the usual way, though their placement rules are the same as for tests - see below - and, once created, they cannot be deleted,only moved.

Tests may or may not have pins associated with them, according to the function of the CASE clause. Tests are initially represented by a # on the nearest internal grid point from the edge of the box plus a spike on the edge pointing towards the #. The spike identifies which edge position is relevant to which # if there is any doubt. A test pin is always an input (solid or hollow, according to whether its TYPE is known) but cannot be named. Tests cannot be placed where ordinary pins would not be allowed. In addition, two tests or a test and a pin cannot lie on the grid points next to the same corner, or opposite each other on a box 2 grid squares wide, because either their #s would then occupy the same grid point within the box or the # and pin name or pin-index would clash.

When the test text is added, the user must decide whether the test is to be in the OF, ELSEOF or ELSE part of the CASE clause. If the user selects ELSEOF he must also give a number defining which ELSEOF arm he means. If an existing ELSEOF number is given, the user is asked if he wants to add the new test to the existing ELSEOF or define a new one with that number, or quit. In the latter case ELSEOF arms with the same and higher numbers are all automatically increased by 1. There must only be one ELSE test. The translator will check that the numbering of the ELSEOF arms is sensible (eg 1, 2 and 4 not allowed) before building a full CASE clause for the compiler.

If the test is valid the # is replaced by an O for an OF test, an E for an ELSE test and the number of the ELSEOF arm for an ELSEOF test, up to 9, after which a * appears which can be interrogated to find the number. We have used an integer-sized, 6-pointed star here to distinguish it clearly from the "*" which is a contracted pin-name, even though the two symbols appear in different positions relative to the pin and/or the edge of the box. The text of the tests can be seen by invoking show_text and clicking on the test symbol itself as this is on a grid point.


ip
ip1
O E
O E
O ch E
O op1
O O
ch op2
E
2 1 2 3 3

5.5) Indexes

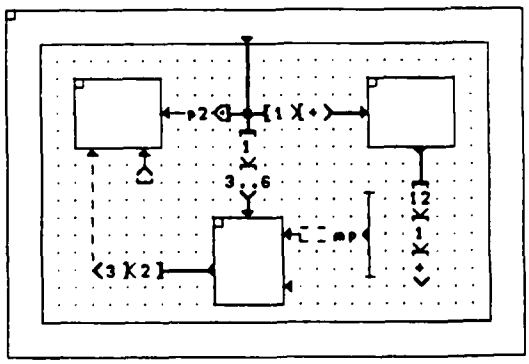
These are also value changing boxes but can never exist in isolation - at least one pin must be joined and therefore the pin TYPE is specified. They each have one input pin, shown as [, and one output pin, shown as > ([and > if their TYPEs are specified), so that the direction of signal flow is obvious from the alignment of the index. Between the pins there can be nothing - an empty index - an integer ([3>), a range of integers ([1..4>), or a composition of these ([3>[2>[1..4>). The compound index still has only two pins to which joins can be made.

The [+> symbol represents a whole index of any complexity which is hidden. The "+" is small enough so that the pins are on adjacent grid points. The [+ > symbol represents the text in one part of an index being hidden, eg [3>[2>[1..4> could be represented as [3>[2>[+ > if space were tight. Here the "+" is obviously larger and occupies its own grid point between the pin symbols like an integer.

Indexes are designed to occupy a minimum of two grid points on the line of the wire they index. They can be accessed via either one or by their text if any and if not completely hidden. Indexes cannot be named but their pins can. Structured wires must always be joined to the input of an index.

Pin-indexes cannot be added to index pins.

Occasionally, moved or inserted indexes will fit in their new positions except for their pin names which, even when fully contracted, overlap with another object. In these cases the index is drawn as a dashed (temporary) box surrounding the pins and must be moved to a legal position before the picture can be compiled.



5.6) Brackets

A bracket is a construct which transforms TYPEs. There is no exact equivalent in text ELLA - the graphical bracket covers REFORM, CONC and indexing. The basic representation for a bracket is a line with pins on each side which forms a very compact construct.

A bracket line can occupy a single grid point (9 pixels centred on the grid point) with one pin on each side representing REFORM. Other forms of restructuring will occupy at least two grid points to allow for three or more pins.

Each bracket has a single input or output (the major pin) for splitting and combining signals respectively, and, eventually, a set of minor pins of the opposite sort for the component signals. The major pin is just an ordinary pin and may be solid or hollow as usual. The minor pins are each represented by a short spike (with or without a cross bar) in the direction of the pin, and therefore covered by the wire when joined. Obviously structured wires must join to the major pin.

A bracket line lies on a grid line but extends beyond the last grid point by 4 pixels in each direction. There is a small tag 7 pixels long across each end of the line.

When a bracket is defined it has no minor pins. The maximum number of minor pins is 2n-1 where n is the number of grid points on the line and the -1 allows for the grid point occupied by the major pin. Two pins (both minor or major and minor) facing in different directions can, therefore, occupy the same point.

Once the TYPE of the major pin is known, either by a join or by specify_type, the minor pins can be placed and have their TYPEs fixed by a default operation - see Section 6.10.5. This gives a pin to each field of the major pin TYPE. The bracket itself is then drawn with its major line double thickness. Done manually it is not necessary to include a pin for every field - only those which are to be used.

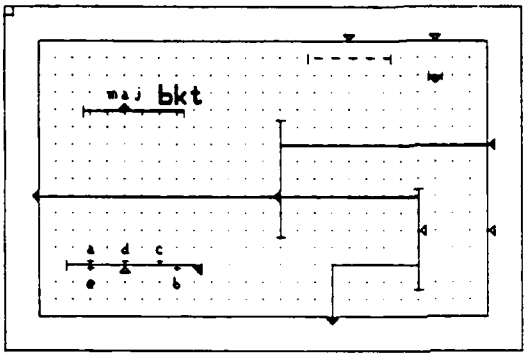
Except where the default pin placing procedure is used or where there is no ambiguity, minor pins which are connected will have to be indexed (using pin-indexes) so that it is known which field of the major pin structure they represent. A bool and an int combining to a (bool,int) is unambiguous, but two bools combining to a [2]bool will need pin-indexes on the pins to show which bool is which.

Brackets can have instance names and pin names but none are compulsory. Instance names (which are local names) are either written next to the bracket line in bold or as a solid triangle at one end of the bracket line. Pin names are written next to the pins, on the same side of the bracket line.

REFORM FNs can, of course, be instanced from declarations already in

the library - in which case they are just treated as ordinary instances with no special syntax.

Occasionally, moved or inserted brackets will fit in their new positions except for their pin names which, even when fully contracted, overlap with another object. In these cases the bracket line is drawn dashed (temporary) without any pins - though wires may still be joined - and must be moved to a legal position before the picture can be compiled.



5.7) Wires

Wires can be simple (one pixel wide) or structured (2 pixels wide) depending on the TYPE of the signal they carry. A simple TYPE is defined by TYPE NEW statements and a structured TYPE is a structure or row of previously defined TYPEs, whose elements may themselves be structures. The structured TYPE may have been defined by a TYPE declaration in the library or it may be a subset or superset of one of these TYPEs created by indexing or bracketing in the new declaration being defined.

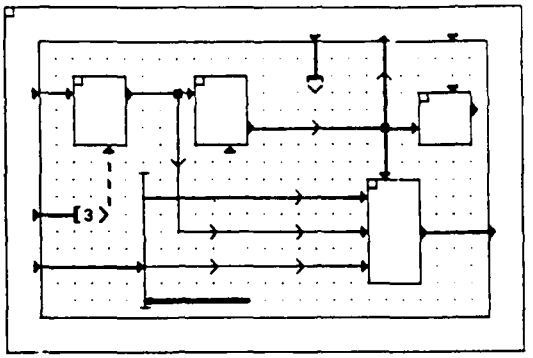
A wire can have any number of segments, where each segment starts and ends on either a pin or a fanout node. Single segment wires always join inputs to outputs. Joins to outer pins (the specification of the declaration) look less obvious than to other pins but with reasonable justification. Inputs to the outer FN are triangular pointing inwards and outputs are triangular pointing outwards, as on instances, but inputs to the FN join to inputs of instances and outputs of the FN join to outputs of instances. This is also true in text ELLA and is much less confusing in use than it seems to be when written down!

Fanout from a node on a wire results in a multi-segment wire with, ultimately, one output, n inputs and m<=n-1 nodes. Each segment may be between an output and a node (one of these per wire), a node and an input (n per wire), or between two nodes. Up to 4 segments may be attached to any one node. The TYPE of signal on a wire is constant over all its segments, ie all the pins must be the same TYPE.

Wires can end in an empty part of the picture, to be picked up later and continued to a valid end. These dangling wires are drawn as dashed lines as they are a temporary structure and, until they are completed, the picture is not valid ELLA and cannot be compiled.

Wires which have been moved because one of their pins has moved may now overlap another ELLA object. These are called temporary and are drawn dashed - either single or double thickness, as appropriate. A temporary wire has to be re-routed on an allowed path or the circuit cannot be compiled.

Wires can have any number of arrows on them to show the direction of flow of the signal. These are not considered when a wireseg is moved so disappear from that wireseg and must be replaced manually if still required - see Section 6.2.6.


{3}

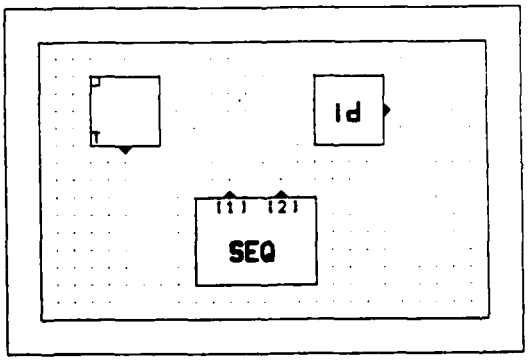
5.8) Text Boxes
Text boxes have a box with no inputs but one or more outputs. Where there is more than one output pin, they must all have pin-indexes and must have been created from a single output field by split_pin.

The size of the box is user-defined though there is a minimum size for which leaves room for one output pin, its pin name, the "FN_name" and the "T" symbol - see below. This box has a perimeter of 6 grid points (a 2x1 rectangle). The box size is set when the Text Box is created but can be changed later so that the pin(s) can be moved for optimum wire routes.

A text box has SEQ, CASE, BEGIN/END or id as its "FN_name" which is displayed or hidden as for local instances (solid square in top-left corner or bold type). There is no instance name.

The text in a text box is always hidden - except for a short "id" it is considered to be too big to try to display. When the text has been added a small "T" appears in the bottom left corner of the box.

The output(s) of a text box must always be joined before the picture can be valid ELLA.



5.9) Names

Names are both important and complex in this system. We hope the necessary complexity will be hidden in the implementation and that nothing about names will seem surprising or inconsistent to the user.

All names can potentially be displayed as their full text in the given font, a user defined abbreviation of that text in the same font, or as a symbol which can be interrogated to find the text behind it. The symbols have been chosen to denote the type of object having that name - for example pin names contract to ▪, the text in an index contracts to ▪ and so on. This method of allowing the user to have names as visible as he likes should allow greater freedom in the placing of components and thus a better chance of displaying the connectivity he wants.

All names are accessible via a grid point within the text or within their symbol except for pin names which occupy space between grid points and are always accessed via the pin itself.

Names never overlap the visible representation (black pixels) of any object except wires where, for clarity, the wire is not drawn in the "box" occupied by the name. Thus, for example, an instance pin name appears inside the instance box where it does not overlap either the pin itself, any of the pixels representing the instance box or any other names inside the box.

It is possible that an object can have a name but it is totally impossible to display it in its current position. There are no invisible names in the system so we always insist that an object is moved to a more suitable position where a name will fit. See Section 6.8.

All names can be changed at any time and all names except compulsory ones can be deleted.

Compulsory names are scoped in the same way as in text ELLA. In addition graphical labels must be unique - for example all pin names in a FN must be different.

5.9.1) Circuit Name

This is compulsory and will be the FN name when the circuit is compiled into the library. The name must be in upper case and can be displayed as full or abbreviated text in the larger font anywhere in the top or bottom border of the picture. If it will not fit it is contracted to a hollow box in the top left of the border area. The FN_name of a circuit being defined locally within another declaration will be in bold TYPE or contracted to a solid box - see Section 7.3.2.

5.9.2) Pin Names

All pins (except CASE test pins) can be named unless they are in one of a few very unusual positions where even the contraction will not fit. Pins are always named in lower case in a small font (7x9 pixels) very close to the pin itself unless there isn't enough room, when a small * (asterisk) is used. The * restricts the pin name to the area between the pin and the next grid point so two adjacent pin names need never overlap. When a new instance is made its formal parameters can be shown as pin names if required.

We use an algorithm which reads in a new pin name, writes it as text or as an * if there is room for either representation, otherwise looks at the existing names (or pin-indexes) in the way and contracts enough of those to *s (or []s) until the new name will fit. It stops the search when an * can be written even if contracting another name, say on the other side, would allow the text of the new name to fit. Operations to contract and expand names and pin-indexes are provided so the user can rearrange text names, *s, pin-indexes and []s as he wishes.

Pins on FN instances, CASE clauses and other boxes have their names within the box but names on outer pins, bracket pins and index pins have to be in the general area of the diagram. If the outer pin name is contracted to an * it is displayed in the border around the circuit area. Brackets can have two pins at the same point. The user is asked which pin he means each time there is a conflict and the name appears on the same side as the relevant pin.

See also pin-indexes (Section 5.10).

5.9.3) Instance Names

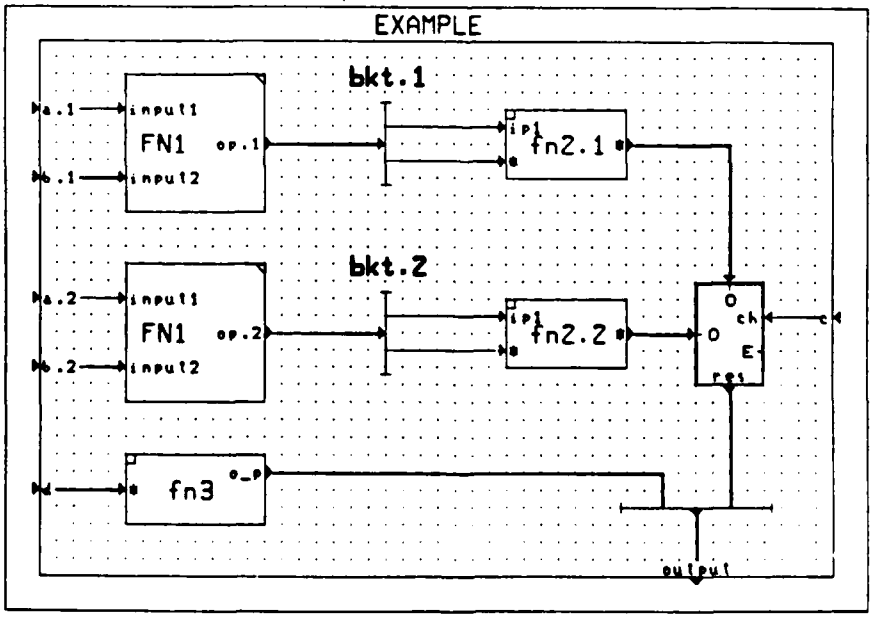
These are mandatory, user-defined names which are written in the larger font in lower case. If possible the text is written in the middle of the box centered on a grid line, otherwise a small hollow triangle is drawn in the top-right corner of the box. The triangle is a way of representing the name without taking up any grid points in the FN or any space in the diagram outside the FN. It is accessed by putting the cursor on that corner to display the hidden name or to pick it up for expansion to text in a user defined place within the FN box. (Hollow triangles are used for instances from the library and solid triangles will be used for local FN instances.) The user can move the name anywhere in the box where he thinks there is room by selecting the position of the left hand edge of the text. The text is centered on a grid point so it can be accessed directly by the cursor. There is also a facility to abbreviate a name to reduce it to a length which is definitely visible. This can be, but need not be, a simple truncation of the original name. Pin names are not automatically contracted to make room for the instance name, though this can be

done manually. Bold type will be used for local FN_names and local instance_names.

5.9.4) FN Names
These appear with the instance when it is made as a hollow or solid box in the top-left corner of the FN box. They are very similar to instance names and are manipulated in the same way. As text they are always written in upper case in the larger font.

5.9.5) Bracket Names
Bracket names are not compulsory. When used they are rather like local instance names and appear in bold type on the grid line next to one end of a bracket, although they can always be repositioned later. Where there is no room for the text, and the user doesn't want a visible abbreviation, a small solid triangle is drawn at one end of the bracket line. There is no corresponding FN name.

5.9.6) Duplicated Names
These are identifed by at least one "." followed by an integer in the name. They can be instance names, outer pin names or names on outputs of instances which must all be unique in their scope. See Sections 6.6.2 and 6.7.

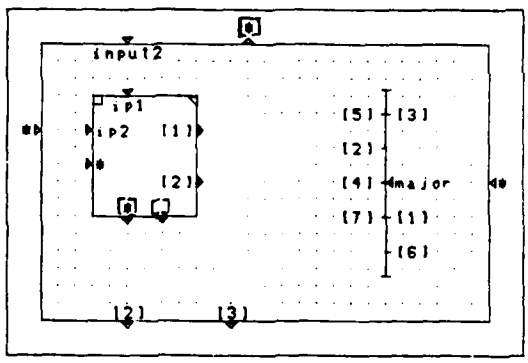


5.18) Pin-indexes
A pin-index is a number in the small font surrounded by small square brackets. Where there is not room for the number and the brackets, the brackets appear alone separated by a few pixels (the [] symbol should take up no more room than the small * for pin names).

Where the pin is both named and pin-indexed there will never be room for both texts or even both symbols. In these cases a compound symbol [*] is used where the whole set takes up fewer pixels than the grid spacing so two named and pin-indexed pins can lie on adjacent grid points.

The algorithm for displaying or hiding pin-indexes is the same as for pin names - in fact pin-indexes are treated as names for the purposes of the naming/pin-indexing algorithm. As for pin names there are a few places where even [] will never fit - these pins cannot be pin-indexed without a move occuring first.

A pin-index cannot be added to a pin on an index.



6) ELLA GRAPHICS SPECIFICATION - PART 3, EDITING OPERATIONS

INTRODUCTION

This section specifies operations within an ELLA picture to produce and edit a valid circuit diagram which can enter the ELLA system as a FN declaration. It complements the previous section which dealt only with the appearance of the pictures.

Checks on the operations tend to be described in two categories where appropriate:- those concerning the graphical shape of an object and those concerning its position, or proposed position, in the diagram which are therefore "ELLA checks".

The operations on the graphical objects can be divided into classes, such as making an object, deleting it, moving it and so on, so the structure of Section 6 broadly mirrors those divisions where possible.

There is a distinction between graphical objects and ELLA objects in this spec as follows. ELLA objects are the graphical equivalents of syntactic elements which exist in text ELLA, eg drawing an instance => MAKE, drawing a wire => JOIN, the outer function => spec of declaration and so on. These objects are stored and manipulated by the ELLA Graphics System. Graphical objects are part of the graphical representation of ELLA objects but have no connection with the ELLA language. For example a box (graphical object) is part of the graphical representation of an instance, CASE etc, or a zigzag line is part of the graphical representation of a wire.

6.1) Making Graphical Objects

In general there are four stages to making any graphical object. 1) Collect points in the picture area, checking interactively that the shape is still going to be valid, 2) check that the graphical object is suitable for its purpose as an ELLA object in this diagram, 3) check that the diagram is clear of other ELLA objects where the new one is to go and 4) add the new object to the data-structure as part of an ELLA object if all the checks have been passed.

6.1.1) Boxes

These are always either part of another object or a temporary structure.

The software should allow the user to define a box by clicking on two points of any diagonal. It should check that there are two points and that they are neither coincident nor collinear on the grid. Error

messages should state the fault and allow the user to carry on (with the current box or a new box, depending on the error) or result.

If the box is OK the software tries to use it in the context intended by the user. If the box is to be part of an ELLA object it checks the size constraints on that object and looks for any intersection between the new box and other objects already in the data-structure. If it passes these checks it is made part of the new ELLA object, added to the data-structure and drawn, otherwise the user is given an error message and offered the option to start a new box or result. If the box is temporary it is part of the delete_area or duplicate_area routines and is allowed to overlap other objects. Temporary boxes are drawn with dashed lines.

6.1.2) Lines

Lines are part of the graphical representation of ELLA wires.

Collect a set of points from one valid end point for a wire to another. Valid end points are pins, wires, brackets or the edge of CASEs, text boxes, or the outer function, where pins are allowed. It is also possible to stop a line on an empty grid point and indicate that it can be picked up and continued later. This is not valid ELLA, though useful graphically, so the dangling wires are drawn dashed as temporary features and the diagram cannot be compiled until the lines are complete

If the first point isn't valid, or any subsequent points are not on either empty grid points or a valid end point, the previous points are lost and the user is given an error message and the option to start again or result.

Line segments are always either horizontal or vertical with one point at each corner of a zigzag. There should never be any lines of arbitrary slope or three points along one grid line. (Selecting points in the order x(1).....x(3)...x(2) on one grid line should have the effect of removing the x(2) from the set of points entirely so that the line is eventually drawn from 1 to 3. Similarly x(1)...x(2)...x(3) removes x(2) from the set as it does not define a corner.)

Marker boxes, continuous drawing or some other technique should be used to show the user the path of the line as he defines it.

In the end, wires must be either between pins or between a pin and a wire. The pin may not actually be present when the line is defined but the pin(s) and the wire can be added together as long as the wire is valid and a deducible sort and TYPE of pin can legally be placed at the relevant end(s). Determining which pin the user means when he joins to a bracket which can have opposing coincident pins is an added

complication but is deducible from the direction of approach of the wire to the bracket.

When a whole set of points between two valid ends has been collected checks on the validity of the wire (as opposed to the line) are begun.

TYPE check the pins or pin/wire at each end of the new wire - see section 6.12.

If all TYPEs are OK, the system tries to add the wire to the ELLA data-structure. It checks the intersection between the new wire and other relevent objects already in the data-structure. If the check fails the points are lost, an error message appears and the user is offered the choice of starting a new wire or pressing result. If passed, the wire is made part of the ELLA data-structure and is drawn. If the wire starts or ends on an existing wire (fanout), a node is drawn at the new fanout point.

6.1.3) Pins

Pins are not graphical objects. They take up no "room" in the diagram as nothing can be placed such that it overlaps a pin - except the wire which joins to it.

6.1.4) Names and Text

Names and text are both graphical objects and attributes of ELLA objects. They are not ELLA objects themselves. Many names in this system are, in fact, graphical labels rather than compulsory ELLA names. The compulsory ones are those which are required to translate the graphical data-structure to text, eg instance names in their own right and as indexable names for output fields where LET names are required, These will appear in the intermediate language representation of the circuit whereas other names will be stored as attributes so that the picture can be redisplayed later as the user designed it.

The user selects a point then the system finds which object the point is on and checks that the object is nameable or can have text associated with it. Nameable objects are FNs (instance name), pins (names and pin-indexes, though indexes cannot also have pin-indexes), brackets and the outer function. Some names can also be selected for renaming as an alternative to selecting the object again - eg a bracket name can be changed by clicking on the old name or on the bracket itself. (Pin names and pin-indexes always live between grid points and therefore can never be selected directly - renaming and re-indexing is done via the pin itself.) Indexes and CASE tests can have text added or existing text edited.

If the point selected is not on a nameable object, give an error

message and wait for the next point or result to be selected. If passed a window about 1x20 characters in size (scrollable along the line) appears near the object for the user to type the text of the name in. If there is already a name it appears in the window to be deleted or edited. The text must not be empty (except under some circumstances for a test text in the ELSE arm of a CASE clause - see Section 6.3.3), and it must be in the correct case with no illegal characters and unique in its scope. These are the checks done by the ELLA compiler on names in text - they should be done interactively in graphics. If the checks fail the appropriate error message is displayed and the user selects another point or result.

If the name, pin-index or text is OK it is added to the data-structure of the object and displayed as text if possible, otherwise as a symbol. The only exception to this is where the user has tried to name or pin-index a pin on a bracket or on an index and there is another object too close to the pin. In this situation a pin name or pin-index will not fit, even as * or [], so the pin cannot be named or pin-indexed without first moving the bracket, the index the pin or the other object. Invisible names or pin-indexes are not allowed. The user must be told that the pin cannot be named or pin-indexed in its present situation and can then continue to name other objects or result to get on with moving the offending objects about.

(N.B. The prototype system at RSRE will demonstrate the name and text software in a much clearer way than a verbal specification can hope to do. We urge the implementors to talk to the graphics team at RSRE to clarify any confusion from the sections on names which have proved more difficult to write than we anticipated. Because, however, we have always followed the rules in text ELLA, we think there is no real confusion and the only genuine difficulty we had in implementing names was spotting the few cases where a name cannot be allowed, as above.)

The symbols for contracted names are given in Section 5. Instance, bracket, FN and circuit name symbols always fit without interacting with any other object. Pin names and pin-indexes check all other names associated with the same ELLA object as themselves, eg within a FN box, to see if they will fit as text or */[]. Often the text of another name will still overlap with the new name even if the new name is contracted. The system should contract as many other names as necessary until the text or */[] will fit. If this final arrangement isn't convenient or pleasing to the user, he can contract, expand and move names or pin-indexes explicitly to see the information he wants.

6.2) Making ELLA Objects

6.2.1) Pins

Static Menu "MAKE Input", "MAKE Output" and "MAKE io".
Display "Input", "Output" or "io".
Select point.
Find which object the point is on.
Legal objects are: outer function
CASE (chooser (input), output and pins for existing tests)
bracket
text box

Pins are not allowed on their own, on wires, in a box, on another pin (except on a bracket where two coincident, but opposing pins are allowed), on the corners of any box or on a CASE box where the name or pin-index of the new pin, when given, would extend over a test already in the box. The pin must also be the right sort, eg the sort of minor pin on a bracket is defined by the major pin, the chooser on a CASE clause must be an input and so on. Pins cannot be added to instances except when the instance is first made from the ELLA library.

When adding a minor pin to a bracket the cursor can be on a point with two pins, one pin or no pins. Obviously a pin cannot be added where there are already two pins. Where there is one pin, a second pin is created facing the other way. Where there is no pin a menu appears asking whether the new pin should be above or below a horizontal bracket or left or right of a vertical one, then the pin is created. These pins are shown by a short spike on the bracket at the pin position and facing in the right direction. When a wire is joined to the pin the spike is no longer visible.

The specify_type procedure can be used on all pins made by the MAKE command, though a join from a pin of known TYPE will also specify the TYPE of the pin at the other end. The representation of the pin changes when its TYPE becomes specified - see section 5.3.

Pins can also be added implicitly by joining to a place where a pin is allowed. Exactly the same rules apply and the wire cannot be made if the pin procedure fails.

If the pin passes its checks it is added to the data-structure of the object it is on and is drawn according to its sort (ip or op), its TYPE specification if known and its object (minor pins on brackets have a different representation from other pins) - see Section 5.3.

6.2.2) Outer Function

The graphics editor is called with the size of the picture as its only parameter. The new picture has a blank two grid wide border with a one pixel wide "frame" round its inner and outer boundaries, and an array of grid points over the rest of the area. The minimum size for

the picture is 6x5 grid squares. This only allows two outer pins a wire between them, pin names and a circuit name but is still a meaningful, if useless, circuit.

Pins are added to the outer function either explicitly as above or by joining a wire to the inner boundary of the border at a point where there isn't already a pin. The pin can be named at any time and the inputs must be named at some time to give the declaration its formal parameters. Specify_type, see below, can be called on a pin at any time.

The circuit name must be added sometime by calling the naming procedure with the cursor somewhere in the border of the picture, including on the inner boundary of the outer function where there is no pin.

The presence of the mandatory names is checked in the translator.

6.2.3) Instances and Imported instances

Static menu "MAKE Instance".

When an instance is requested a window appears containing a scrollable menu of all the declarations in the context being used. There is also the option in the menu of changing the context to look at a different set of declarations which are the exports of the new context. This does not change the context the user has asked to work in, but it allows him to IMPORT declarations from other contexts. The user clicks on the one he wants, the window closes and there is a display outside the picture area of "FN fn_name = ..., define box", where the whole spec is shown so the user can still see how many pins there will be on the box.

Alternatively, if the graphics system is running within ELLAview, the user may point at the icon representing the declaration he wants in an open context window and the information associated with that declaration is then brought into the graphics system, together with context information if the declaration was not in the working context. The display appears as above. (See next paragraph.)

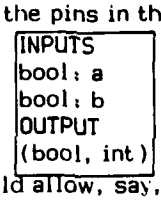
(NOTE: This is an alternative mode of working as the first paragraph above specifies a means of making an instance which is possible in any graphics window whether it is generated within ELLAview or not. In addition, the independent method gives the user the specifications of all the available declarations which he may find useful, whereas ELLAview just gives FN_names. For this reason it should not be assumed that the user will want to use ELLAview icons to MAKE new instances in his picture - instead, when the graphics window is within ELLAview there should be a toggle switching between the two

modes of operation at any time the static menu is accessible to the cursor.)

The user defines a box (Section 6.1.1) which must be big enough for the total number of pins in the FN spec. If this check fails the user can continue to try other coordinates and positions or result/quit. If OK the box is drawn. An imported FN has a small "I" in its bottom left hand corner. This can be interrogated, using the bottom left corner of the box for access, to find the context it came from. If the imported FN_name clashes with a name in the main context the user must rename the import. The new name becomes the FN_name of the import which is then treated the same as any other FN name. Of course, the original name must be kept with the original context information so that the circuit can be translated to correct ELLA using the RENAME facility.

Display "FN fn_name = (spec as in library)"

A scrollable window of constant size, not overlapping the new box, appears containing a list of the pins in the spec, eg



The size of the window should allow, say, 6 or 7 pins to be visible at once without scrolling which makes it dependent on the font size of the host machine.

The user can put the cursor on any line, click and return to the picture to place the pin which must be in a valid position on the right box. If the pin is successfully placed make_pin, specify_type and, if an input, the naming procedure are called implicitly - the name being the formal parameter at this stage - and the line in the menu is marked as being done, eg by reverse video. This continues until all the pins are placed. Split_pin can be called to separate the fields of the output once it has been placed, if necessary - or the menu reader could include a way of differentiating between a single pin for the output and a pin for each field. In the latter case pin-indexes will be necessary. (Brackets can be used to restructure any pin TYPE outside the instance.)

Of course a linear representation of the spec in a scrollable linear window and a more sophisticated menu reading program could be used here. I am specifying that the user should be able to point to some representation of the FN spec and place the pins from there without typing any text.

There should also be the option of asking the system to make an

attempt at placing the pins, eg if there are 100s of them. There are many possible ways of doing this for textual or graphical declarations, all of which are equally unlikely to be right for the connectivity of the current diagram. A simple scheme should be adopted for a FN defined textually, eg start placing pins clockwise from a given corner (bottom right - so that most inputs end up on the left and top of the instance). Or a graphical declaration whose box sides were big enough could have the original pin positions scaled to fit on the same side of the instance box. Of course, the user can always move any pin using move_object later.

It is important that ALL pins should be placed before the menu disappears. The user is making an instance and does not have the option of leaving pins off. All pins on an instance have pins of known TYPE and are, therefore, drawn as solid triangles.

6.2.3.1) Names

An instance name has to be given sometime. It could be required now, an ignorable prompt could be given, or it could be left to the translator to check that the user has done all the instance names. The last option is probably sufficient.

The FN name first appears contracted but can be expanded at any time. A call of show_formal_parameters on the instance will add the formal parameters of the declaration as input pin names if required, and if not done automatically.

6.2.4) CASEs

Static menu "MAKE CASE".

Display "Define CASE box"

The user defines a box as above which must be big enough for the two real pins and at least two tests. If the checks fail the user can continue to try other coordinates and positions or key result. If OK the box is drawn with double thickness lines and the CASE is added to the data-structure.

Tests on CASE clauses may or may not have pins associated with them - it depends on the function of the CASE clause. The tests are added either by joining to a place where a test is allowed or by make_test. (Static menu "MAKE Test".) Tests are allowed in the usual pin positions, except where the #s of two tests would be coincident. (This only matters on narrow boxes - two grid squares wide - or on the grid points next to corners.) A click on "Add Test" must be followed by a click on a valid position on the edge of a CASE box. If it is valid a small spike pointing into the box from the cursor position and a # symbol on the internal grid point are drawn. The spike

identifies which edge position the test refers to. If the test is added by a join to the edge of the CASE box an input pin is drawn as well as the other symbols. Make_input can be called to add an input pin to an existing test as long as the test text is not yet added. Test pins may be added to tests implicitly, by joining to an existing pinless test which has not had text added. The system checks that the test text is absent and the user is then either told a pin cannot be added, if text is present, or is offered the choice of adding the pin or quitting.

The chooser and output are added as normal pins at any time - ie implicitly by joining, or explicitly by make_pin with or without specify_type. These pins can be placed anywhere on the box where a test can go - see above.

The usual checks if the TYPE or sort of pin in an implicit make_pin are ambiguous are carried out - for example a join from an output to a CASE must add an input to the CASE. If there is no chooser, the user must be asked whether the new pin is to be the chooser or a test pin. If there is a chooser the new pin is obviously a test pin. The CASE clause must have only one output, of course. The TYPE on the wire must be compatible with the TYPEs already known on the CASE pins.

Specify_type can be used on all pins. The TYPEs of the different sorts of pins on a CASE are linked. Test pins are the same TYPE as the output and the test texts must all be basic values such that their TYPEs are "input TYPE: output TYPE", "input TYPE:", "output TYPE" or "" depending on the sort of the test and whether or not it has a pin. Thus specify_type on a CASE pin may set more TYPEs, and redraw more pins, than the one selected and joining or adding test text may also specify some pin TYPEs. The presence or absence of a pin and the selected arm of the CASE clause will determine the form of the test text. The CASE clause must be consistent at all times.

Split_pin can be used on the input and output once their TYPEs are specified (when pin-indexes will become necessary) though they must first be made as single pins as usual.

6.2.5) Brackets

Static Menu "MAKE Bracket - Split Signal
Combine Signals"

(If the Split Signal option is chosen, the major pin will be an input and the minor pins will be outputs. The opposite arrangement is true for the Combine Signals option.)

The user selects two points on a grid line which may be coincident (but he must click twice) and they are shown by marker boxes or a line.

If the points define a valid line, the user is immediately asked to select a position and indicate a direction for the major pin. In the prototype this is done by selecting a point between the ends of the line but offset from it. The major pin position is the grid point on the bracket line closest to, and pointing towards, the offset point. If a valid major pin position has been selected the bracket line and major pin are drawn.

If the points are coincident a third point (not coincident) is used to set the direction of the major pin and, hence, the direction of the bracket itself. The third point must lie on the horizontal or vertical grid line defined by the position of the two coincident points. The bracket line is drawn centered on the coincident points and the major pin is drawn on that line pointing towards the third point.

The procedure will fail if two points defining the basic shape of the bracket are not set or if the major pin position does not satisfactorily define its position and direction on the line. If the bracket is graphically well-defined it is checked for position against other objects in the picture then added to the data-structure. As usual, relevant error messages and the option to result or carry on are given where necessary.

Minor pins are added using Add Input/Output or by *joining as usual*. They must be the opposite sort to the major pin. They are represented as short spikes (with a cross bar if their TYPE is not known), less than half the grid spacing in length - a notation which is particularly clear as it prevents clutter on bracket lines where pins with opposite directions can occupy the same point.

6.2.6) Wires

Static Menu "MAKE Wire".
A wire has either a single segment, from input to output (or equivalent) or many segments which take the signal from one output to many inputs via fanout nodes. A segment may, therefore, be 1) output -> input, 2) output -> node, 3) node -> input or 4) node -> node.

All wires start off as simple one segment wires between an output and an input - though it may be defined in the other direction. Further segments are added by drawing lines from inputs to wires, inputs to nodes or vice versa. A node is drawn wherever fanout occurs and two or three wire segments can "leave" each node. Node -> node segments are not drawn explicitly, but are made as the user fans out from a segment of wire which itself ends in a node.

The TYPE on all segments of any wire is always constant and is always known. A wire cannot be drawn between two points of unknown TYPE. If the TYPEs at both ends are known and are equal the wire is allowed, if

they are different it is not. If only one TYPE is known the wire is allowed and the unknown TYPE is set automatically. An index, therefore, marks the end of a wire as it changes the TYPE of the signal passing through its "box" and it cannot be made unless the TYPE at the start of the wire was known.

A wire should not have any closed loops (short circuits) in its representation.

The user can stop drawing a wire at some empty point in the diagram and continue it later. While he is collecting points for the line he can press result and is asked whether he wants to make an index, leave the wire dangling or quit. If he chooses the second option, the wire is drawn dashed (temporary) to that point and can be picked up later by selecting its end point as the first point of a "new" wire and continuing as usual to another valid end (including another dangling point). The two sections are then combined in the data structure to a single wire or wireseg and the dashed line is replaced by a solid one. A dangling wire can only be continued or deleted. The TYPE carried by a dangling wire need not be known, though it must be set at or deducible from the other valid end point when the wire is completed. If the TYPE remains unknown, the continuation is not allowed but the original dangling section is not lost. For the purposes of the intersection routines a dangling wire, including its end point, is a valid graphical object and cannot be overlapped.

Arrows can be drawn on a wire at any grid point, static menu "WIRE Arrow", except a corner and a node, to show the direction of flow of the signals. The user can keep on adding arrows to any wire until he presses result. The system calculates the direction of the arrow in each case and draws it automatically. The user can fanout from an arrow but the arrow disappears. Arrows do not move when a wireseg is moved - they must be redrawn if still required. (It is unlikely that any automatic placement of arrows on wires would be satisfactory to the user so we insist that they are replaced manually on the mobile wiresegs.) As the arrows are not ELLA objects, they cannot be deleted by delete_object. A delete_arrow procedure deletes each arrow individually, or all arrows on a wire segment, or all arrows on an entire wire - one of these options picked by the user from a menu. The delete_arrow procedure stays in the chosen mode until result is pressed, when the menu reappears and the user can select another mode or result again to leave the procedure.

6.2.7) Indexes

Indexes cannot be made in isolation and no sequence of events should leave an isolated index in the picture.

As part of defining a wire the user can select a grid point in a clear area of the picture. This is usually a corner of the wire and the user

goes on to end the wire elsewhere. If, however, he presses result with the cursor on a grid point he is offered the choice of making an index, leaving the wire dangling or quitting. If he chooses to make an index the procedure checks that the wire is allowed, the TYPE at the start of the wire is known and the next grid point along the line is clear, then the symbol [> is drawn along the line of the wire with either [or > touching the wire to denote the direction of flow of the signal. In other words, [is an input pin and > is an output pin, each pin being on a grid point. The pin touching the wire is of specified TYPE and is, therefore, drawn in bold.

If there is not room for the [> symbol the previous points are lost and the user must start a new line or result to do something else.

Once the [> symbol has been drawn the user, who is still making wires, can start a new one or result as usual. The second wire from the index can be drawn at any time, not necessarily before the text is written in.

The text in the index is inserted by the text procedure called on the index (See section 6.3.2.1).

Specify_type can be called on unjoined index pins.

6.2.8) IMPORTs

Static Menu: "MAKE IMPORT".

A method of making IMPORTed instances whose original context is known is given in Section 6.2.3. However, it is also possible to make a box with pins and link it with an instance from another context later.

The user clicks on MAKE IMPORT then defines a box as usual in the picture area. The box is drawn with a small "?" in its bottom left corner. He is asked immediately to give a FN_name which must be unique in the working context. The IMPORT needs an instance-name as usual. The user can add pins to the box explicitly, with or without specify_type, or by joining. All pin TYPEs must be specified before the picture can be compiled. Where there is more than one output, pin-indexes must be supplied. Where an input pin is added a name must also be given - these names will become the formal parameters of the input so will be used later with the FN_name when the spec of the IMPORT is matched to declarations imported from elsewhere.

A picture in this state can be compiled but not simulated. (It is equivalent to a text file containing, for example, "FN NOT = (bool: ip) -> bool: IMPORT." outside the current declaration and "MAKE NOT: not." inside it.) The usual text statement, "IMPORTS other_context: FN_name" or IMPORTS other_context: OTHER_FN_name RENAMED

FN_name" must be compiled into the context first. Once these statements have been added to the context and the system has checked that the specifications match, the data-structure is updated with the importing context information and subsequent displays of the declaration show the IMPORT with a "I" in its bottom left corner. If there is a mis-match in names, formals or number of pins the picture still cannot be simulated and the closure of the context remains incomplete. The closure information from the EASE must be used to show where the inconsistency lies, as in text declarations.

(It is possible that a give_import_context procedure could be implemented, though it requires a sophisticated interface with the EASE. The user clicks on an empty IMPORT and is prompted to type in a text window "another context name/FN_name in that context" (if RENAMED is to be used). The system must then "compile" this, together with the FN_name given to the IMPORT when it was made, into the ELLA library as if it came from a standard text file, add the information to the graphical data-structure and redraw the "?" as "I".)

6.2.9) Text Boxes
Static menu: "MAKE Text Box SEQ|BEGIN/END|CASE|id".

The user defines a box in the usual way. The box must be at least 2x1 grid squares in size - see Section 5.8. The word defining the sort of Text Box - "SEQ", "BEGIN/END", "CASE" or "id" - as chosen from the Static Menu is treated like a local FN_name and is automatically written in bold or hidden behind a solid square in the top-left of the box.

A picture containing a text box cannot be compiled until the text is filled in and the output is joined - see Section 6.3.4.

6.3) Text
6.3.1) Names
6.3.1.1) Adding a Name
Static Menu - "TEXT Name".
Names are not strictly ELLA objects - they are attributes of ELLA objects. However, the rules for making names are similar to the rules for making genuine ELLA objects. Names also follow the conventions of their equivalents in text ELLA. The case and characters used for each name must be appropriate for the object being named and the new name must be unique in its scope in the same way as are text names. These points are checked before the name is allowed and error messages given if necessary.

Nameable objects:
FN - instance name is mandatory. Once given it can only be edited by a further call of the naming procedure - not deleted.

Bracket - name is instance name but is not compulsory. An empty grid point with no minor pins will select the bracket, as will the major pin position where choices are given as in the pin section below. The text of a bracket name is in bold type and its symbol is solid because it is a local name.

Pin - Inputs to the outer function must be named (if used). These names can only be edited, not deleted, as for instance names.

Some pins are in positions where even * will not fit, eg where a bracket is on the next grid line to another object, where an index pin is next to another name in the general picture area so that the index pin name would overlap the other name, or in the outer function where there is no room for the text and the * would clash with the circuit name. Either the circuit name, the pin, the bracket, the index or the nearby object will have to be moved or contracted before a name can be given to the pin, and a warning message to do something is given. There are no invisible names. (These situations are likely to be rare.)

Where there is ambiguity about the object to be named a menu is used to give the choices. This is most apparent on brackets where two pins can occupy the same grid point and the selections are "Name upper pin"/"Name lower pin" for horizontal brackets and similar choices for vertical ones. In addition the major pin may be coincident with a minor pin and in fact the major pin position may be the only way of selecting the bracket if all the grid points have minor pins on them. These two possibilities lead to two menu selections - "Name Bracket"/"Name major pin"/"Name minor pin" or just "Name Bracket"/"Name major pin".

Outer function - name is circuit name and is compulsory. A point in the border around the picture, or on its boundary is selected. The system tries to place the name centrally in the top border or centrally in the bottom border. If these places are both obstructed the name is automatically hidden in a small hollow box in the top-left corner of the picture. The circuit name can never be placed in the left or right borders of the picture.

Name - The name procedure on an existing name (if it is selectable) or the named object is used to edit that name. Blank names cannot be entered - delete_text must be used to remove a name and will only work on non-compulsory names. The checks on case, character set and uniqueness of name in the relevant scope are carried out each time.

6.3.1.2) Deleting a name
Static Menu - TEXT delete.
Delete on a name removes it entirely as long as it is not a compulsory name. The name is accessed by putting the cursor on the text or the symbol for all names except pin names which are accessed via the pin position as they live between grid points. Names which are compulsory cannot be deleted - a warning message should be given and no change is made.

6.3.1.3) Expanding a name and abbreviating
Static menu "TEXT Expand".
Works on all names - in full, abbreviated or hidden. The user is prompted to select the name to be expanded and receives a warning if the cursor is not on a name. If it is and the name is a pin name, it is written in automatically if it fits as there is no choice of position for pin names. For other names, he is asked to select the left edge position for the expanded name and it is written in if it fits. If the name is too big for the compulsory or user-defined position, the user is offered the option abbreviating or quitting (pin names) or trying again in another position, abbreviating or quitting (other names). If he chooses to abbreviate a window appears with the full name in it which can be edited as usual. The abbreviation can be any combination of characters usually allowed in a name of that sort and is not restricted to being a truncation of the full name. The abbreviation is not checked for uniqueness in its scope - that is up to the user - though in other respects the name should be legal.

A call of expand_text can be used to move a name (except a pin name) by selecting a different position for it.

6.3.1.4) Contracting a name
Static menu "TEXT contract".
The user selects any name by pointing at the text or pin position as appropriate. If the name is abbreviated already the name is hidden as required and the abbreviation is lost. (Abbreviations are only provided to make long names visible.) If the name is already as small as possible an error message is given, otherwise the design of the name symbols is such that the procedure will always work.

6.3.2) Index Text
6.3.2.1) Adding Index Text
Static Menu - "TEXT index".
(N.B. Section 5.5 shows indexes in a diagram. It is clear which "+"s are small and between grid points and which are large and occupy their own grid point. The printer used to produce this document has no "+" symbols which are obviously different in size and do not look

as if I mean them to be in bold type. Wherever I use "+" in the context of a contracted index or index part I will state whether it is small or large. Small "+"s never occupy a grid point and only appear surrounded by a single set of brackets as the index, whatever its complexity, has been fully contracted. Large "+"s never fall between grid points and only appear in indexes with more than one visible part.)

The cursor must be either on one of the pins, if the index is in its most compact form or empty, or on the text itself if already present. A window appears in which the user can write the text.

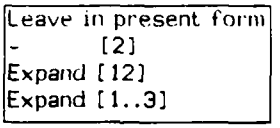
This must be in the form of a genuine ELLA index, eg [2][12][1..3], and checks are made on brackets (square only and right number), ranges (LHS < RHS) and characters (only numbers and "."s) as in the compiler. If the text is unsuitable the user can try again, by editing the display, or quit. Graphically, as shown in Section 5, the ELLA text brackets are transformed to [2>[12>[1..3> to show the direction of flow. Each integer or range of integers is called an index part - thus [2>[12>[1..3> has 3 parts.

The TYPE checker is called before any graphical placement is attempted. The TYPE on the input must be indexed by the text to deliver the TYPE on the output. As much checking as possible on actual known TYPEs, structured or simple TYPEs or just the number of fields in a TYPE should be done and if the procedure doesn't actually fail the text is assumed to be OK. Addition of text to an index whose input TYPE is specified serves to specify the output TYPE - the pin is redrawn accordingly. One pin TYPE on the index is always known as the index cannot exist without being joined to something else.

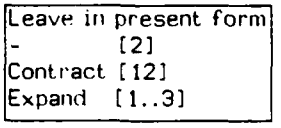
The system then tries to insert the whole text, moving the output pin and keeping the input pin stationary. (A convention.) If this fails an attempt to move the input end is made and if there is still no room for any expansion a small "+" is placed between the pins, neither of which move. This is the most compact form an index can take.

There may, however, be room for some but not all of the expansion because the index, or some of its text in the vertical case, overlaps another object. For a vertical index any part of the text which cannot be shown because the text itself overlaps another object is contracted to a [+>, (large "+"). This is never necessary for single digit text, only for multiple digits and ranges, eg [12] or [1..3]. The "height" of a vertical index, in grid points, is always 2n+1, where n is the number of parts. The "length" of a horizontal index, however, is determined by the text, so it is possible to contract individual parts to make the whole index a minimum length of 2n+1 as above. In this case the index is drawn with all multiple character parts as [+> (large "+", eg, [2>[+>[+>) and a menu appears showing

what the user can change if he can see an expansion that would fit. Using the example above, this would look like:



If the user chooses to expand the [12] he will be asked which end of the index to move and if the expansion is possible it will be displayed (as [2>[12>[+>) together with the following menu:



The user may now try to expand the [1..3]. By carefully choosing the end of the index to move it may be possible to fit the complete index ([2>[12>[1..3>) but, if not, the [12] could be contracted again to make room for the [1..3] to be expanded to give [2>[+>[1..3>. The "-" means there is nothing to be gained by changing the 2 since it takes up the same space as a large "+". When a selected expansion is not possible a warning is given and the same menu is displayed. The process stops when the user selects the "Leave in present form" option.

The text procedure called on an index whose text already exists will give the user a window showing the existing text which he can then edit. The new text is treated as totally new, so the positioning routines, TYPE checking routines and so on are all invoked. The text cannot be edited to "" - this must be done using delete_text.

6.3.2.2) Deleting Index Text

Static Menu "TEXT delete".
Individual parts of an index cannot have their text deleted - the index text can only be edited or completely deleted.

To delete the whole text the user puts the cursor on the text itself or on one of the pins if the index is fully contracted. The input pin is kept fixed when the contraction takes place and the result is the empty index symbol, [>.

6.3.2.3) Contracting Index Text

If the cursor is placed on the text of part of an index which can be replaced by [+> (large "+") this will be done. For a horizontal index the user must be asked which end is to move.

If the cursor is placed on one of the pins the index will be reduced to compact form, moving the selected pin.

Selecting any other position in the index, eg the >[grid points if any, will produce an error message and the user must then try again.

6.3.2.4) Expanding Index Text
For a compact index the expand procedure will try to display the full index text, moving the end selected by the cursor. The same process is followed as when adding text, see Section 6.3.2.1.

If the text is partly expanded the cursor must be on a large "+" symbol and the system tries to expand just that part. Selecting any other part of the index will result in an error message and the option to try again. For a horizontal index the user must be asked which end to move.

6.3.3) CASE Test Text
6.3.3.1) Adding CASE Test Text
Static Menu - "TEXT test OF|ELSEOF|ELSE"
The text of a test is added by a click on the static menu followed by a click on a #. A window appears near the test in which to write the text as it would appear in ELLA text. When the user has typed in his test text the window closes.

If ELSEOF has been requested a window is needed in which to enter an integer for the number of the ELSEOF arm in the CASE clause. These numbers are checked in the translator for gaps and they will be renumbered consecutively before the circuit will compile.

If the user already has an ELSEOF arm with a given number and enters the same number for some new text, he either wants that text added to the existing ELSEOF or he has made a mistake somewhere. He should be offered the choice, by menu or mouse keys, to add to the existing ELSEOF, to make a new ELSEOF with the given number (and therefore automatically renumber and redisplay other arms with the same and higher numbers) or Quit.

Once the sort of test has been determined the checking of the test text begins. First, it is TYPE checked for consistency with TYPEs already present eg, on chooser, output, other tests or input pin of this test, whichever are already present. The new test text may also be able to set some previously unknown TYPEs - redrawing of pins being done automatically and the deduced TYPEs added to the data-structure.

Secondly, the form of the text is checked against the presence of a

test pin and the sort of test the user has requested. For example, "(t, bool):t" could be the text of an OF or ELSEOF test with no pin, "(t,f):" could be the text of an OF or ELSEOF test with a pin, "t" could be the text of an ELSE test with no pin or, finally, " " could be the text of an ELSE arm with a pin. The tests with pins are the graphical equivalent of JOIN CASE... or nested CASE clauses in text and those without pins deliver constants. We assume that the user was right first time, that is "(t,bool):t" cannot be the text for a test with a pin and "(t,f):" cannot be the text for a test without a pin. The test pin is not added implicitly - this may be the wrong decision but is easily changed if the users object.

Finally, the text is checked against other text in the CASE arms for disjointness as in the text compiler.

An error message appears if the text is not of the correct TYPE(s), form or disjoint and the text window is reopened for the user to edit the text or quit.

When the text is completely correct the # is changed to "0" for OF, "E" for ELSE, the numbers "1..9" for ELSEOFs 1..9 and "*" for ELSEOFs with higher numbers. These symbols are centered on the internal grid points and the spike remains pointing to them. The "*" can be interrogated to find its number. As specified in Section 5.4, this "*" is visually distinct from the "*" used for pin names.

Test text can be edited by recalling the "TEXT test..." procedure with the cursor on the test symbol. This may, of course, change the sort of test as well as its text. The text cannot be edited to "" in this way, except for the text of a new ELSE arm with a pin in order to allow the JOIN CASE and nested CASE constructs.

6.3.3.2) Deleting CASE Test text

Static Menu: "TEXT delete".
The user clicks on a test symbol and it is immediately replaced by # as both the text and the sort of test are lost. A click on a # does nothing except issue a warning that there is no text to be deleted. Test pins, if any, are unaffected.

6.3.4) Text Box text

Static Menu: "TEXT Text box".

The user clicks on a text box and is given a scrollable window in which to type ELLA text.

6.3.4.1) ID Box text

In the case of an ID box the window is the same as a text window and the user is prompted to write an existing name (not a graphical label) in the window. The name must have been defined already and must be in scope at the same level of description as the text box. This is checked when the user presses result to close the window. An error message is given if the name is invalid and the user can select the same text box again for another try. It is not possible to type a new name into the ID box to be checked later for an identical name in the same scope.

(The ID box effectively brings a wire from the point with the name outside the box to the output of the text box - a black hole, or two layer wiring. An upgrade to this system would be to allow a wire to end at an input to an ID box, as well as appearing at an output, which would be closer to a possible real layout of the circuit though not electrically different from the present specification.)

6.3.4.2) Other Text Box texts

In the other cases the window is much larger - the equivalent of an empty text file - and already contains the words CASE, BEGIN SEQ or BEGIN on the first line and ESAC, END or END on the third (last) line according to the type of the text box which is set when it is made. The cursor is on the second line of the window and cannot be moved to the last line. Further lines of text are always above the final language word.

The user types his ELLA text exactly as he would in a text file, including other Language words such as OF, ELSE, OUTPUT. He has available all ELLA names (not graphical labels) in scope at the same level as the text box. The names he uses act as the "inputs" to the text box. Other names declared within the Text Box must be unique in their scope as in ELLA text.

When he has finished he presses result to close the window. The system must then compile the new text, for example by enclosing it in a dummy function whose input is made up of the names in the picture used in the text and the TYPEs of the objects with those names, and whose output is the output unit of the text. If all the text is valid the window closes, otherwise it remains open with the cursor on the first error. The user must be able to step through the errors to correct them all before pressing result to try the checks again, or he can press quit which looses the whole text.

This compilation must deliver the TYPE of the output unit of the text which then sets the TYPE of the output pin on the text box if this is not already known. If, however, there is a mis-match the window is left open with an error message giving the TYPEs to show the inconsistency. The user is then offered the choice of quitting, editing the text or changing the pin TYPE. If he chooses the latter,

the window closes and specify_type is called internally to reset the pin TYPE to the TYPE of the output unit.

(N.B. The exact method of compiling the contents of these Text Boxes interactively is not specified, nor is the process by which a text window is provided for the user - these are implementation decisions. The dummy function method is simply one possible approach to interactive compilation. The Text Boxes should not transform to dummy functions when the picture is stored in the EASE as this would lead to declarations appearing which the user never explicitly made.)

A Text Box containing valid ELLA text has a small "T" in its bottom right corner. Regardless of the presence of an output pin on the box, the Text Box knows its specification from the text so the output pin TYPE is automatically specified if the pin is added after the text.

The text can be edited by a further call of TEXT text box - the window then appears with the original text in it and can be edited like a text file. The above checks are re-done when the user tries to close the window.

6.3.5) Pin-indexes

6.3.5.1) Adding Pin-index Text

Static Menu: "TEXT pin-index".
A pin-index is made and operated on exactly like a pin name, though all pins except index pins can have both. The procedure to add a pin-index to a pin will be called both explicitly by the user and implicitly by procedures which manipulate pins and TYPEs in certain ways. The user must be asked which field of the compound TYPE this pin's TYPE represents. An integer in []s written in a text window is sufficient information. If the new integer is already present the user is asked whether to edit it, keep it and renumber all higher pin-indexes (if applicable to this pin) or quit. If he renumbers, the data-structure and display are changed automatically. This will only be allowed if the TYPE checker passes the new pin-indexes as valid.

Where the pin is unnamed, the new pin-index is displayed as [integer] if other names/pin-indexes in its area (eg inside a box) allow it, otherwise as []. Where the pin is named both texts are contracted to the compound symbol [*].

Some pins are in positions where even [] will not fit as for pin names and the same action must be taken - section 6.3.1.1. Where there is ambiguity about the pin to be pin-indexed a menu is used to give the choices. This is most apparent on brackets where two pins can occupy the same grid point and the selections are "Pin-index upper pin" / "Pin-index lower pin" for horizontal brackets and similar choices for

vertical ones. The major pin may be coincident with a minor pin but there is no ambiguity here as major pins cannot be pin-indexed.

Pin-indexes are compulsory on outputs from the outer function but are only required elsewhere when the TYPEs are such that confusion can arise. (Where pin-indexes are not compulsory, pin names can be used as labels to identify pin TYPEs if necessary.)

Pin-indexes are not operators which change the value of a signal. The index on a wire construction must be used for that.

6.3.5.2) Deleting Pin-index Text

Static Menu "TEXT delete".
A pin-index can be deleted though the system must spot if this leaves the TYPEs on an object in an ambiguous state. Such a state is then flagged and the picture cannot be compiled until the necessary pin-indexes are added.

If a pin is both named and pin-indexed a menu appears asking the user which he wants to delete.

6.3.5.3) Contracting Pin-index Text

A pin-indexed pin is selected and the text is replaced by [ ]. If the pin is both named and pin-indexed both are already displayed as [*].

6.3.5.4) Expanding Pin-index Text

Static menu "TEXT Expand".
Works on all pin-indexes on unnamed pins. The user selects such a pin whose pin-index is then written as [integer] if there is room, otherwise there is no change. Pin-indexes cannot be abbreviated.

6.3.6) Show text

The user selects any full, contracted or abbreviated name, any CASE test (cursor on the test symbol, not the pin) or any full or contracted index text - either a fully contracted index or a contracted part - and the full text behind the symbol or abbreviation or the full index text appears in the display area outside the picture.

If there are two or more texts available from one point, eg 3 names from the major pin position on a bracket, a pin name and index text on an index or a pin name and a pin-index on a pin, all the texts should be shown with the appropriate labels.

6.4) The stack
The stack exists to hold some types of object and groups of all objects deleted or duplicated from the current picture. It works on a last-in-first-out basis. The basic objects it can hold are FN instances, CASEs, Brackets and text boxes. The groups of objects can also include whole legal wires, outer pins and indexes - this is covered in the sections on delete_area and duplicate_area (6.5.2 and 6.6.2).

The top element on the stack can be inserted anywhere on the diagram where there is room. If the insertion fails the user can quit or try again elsewhere. Quit puts the object back on the stack - it must not be thrown away or the user is not given the chance to rearrange other objects in the picture to make room for the new one.

An explicit operation "remove from stack" must be provided to delete objects from the system altogether or the stack will get jammed. This could have an integer parameter to remove the top n elements.

Operations to "show" all or the top n elements of the stack and to insert the nth in the diagram may well be very useful. The prototype has a very primitive stack which stores objects as specified above but only inserts the top one. (This is, in fact, sufficient as the user can, as a last resort, enlarge the entire picture area, insert the top n elements and re-delete them after manipulation.) More experience with real designs in circuits where space is tight is needed to specify a necessary and sufficient set of user-friendly operations for objects temporarily out of the picture.

6.5) Delete Object
6.5.1) Deleting individual objects
Static menu "DELETE Object"
This is a context sensitive command.

Deletable Objects:
Pin - not added to stack. On a bracket, the choice of left/right or upper/lower minor pins is given. If the major pin position is selected the minor pin coincident with it is deleted (if there is one) or the choice "Delete Bracket"/"Quit" is given. Pins on instances, the chooser and output on a CASE clause, the output on a text box, the major pin on a bracket and pins which are fields of a split pin (Section 6.10.4) cannot be deleted - only moved. Pins on indexes cannot be deleted and can only move when the whole index is moved. Pins on CASE tests can only be deleted if the test is still unset (#). Otherwise the whole test can be deleted or delete_text used to unset

the test (symbol replaced by #), then the pin can be deleted. Pins must be unconnected before deletion.

FN instance - can only be deleted if not connected to anything. Added to stack.

Bracket - can only be deleted if not connected to anything. Added to stack. Accessed either directly via point on line with no pins, or indirectly via major pin position and menu.

CASE - can only be deleted if not connected to anything. Added to stack.

Text Box - can only be deleted if not connected to anything. Added to stack.

Wire - not added to the stack. Wires are accessed by the cursor on any point of a wire except a fanout node. The pin position at an end of a wire will therefore access the wire - once the wire has gone the pin can be deleted from the same point. If the wire has a pin at both ends the whole wire is deleted. If it is a complex wire it has various segments - one from an output to a node {1}, two or more from nodes to inputs {2} and possibly some from node to node {3}. If the cursor is on a segment of type {1} the whole wire (every segment) is deleted, if on type {2} the node to input segment is deleted, if on type {3} nothing is done and an error message is given as this sort of segment cannot be deleted on its own. Deleting a type {2} segment will result in the node being deleted as well if there were only two segments leaving the node. If the wire has an index at its end and the other pin of the index is not connected, the index is deleted as well. A dangling wire can be deleted.

Index - not deleted directly. The text is removed by delete_text to leave the empty index symbol [>. The index symbol with or without its text is removed by deleting the last wire connected to it. This provides a way of completely removing the index from the diagram without it ever existing in isolation which would be meaningless in ELLA. Indexes are not, of course, added to the stack.

6.5.2) Deleting groups of objects

Static Menu "DELETE Area"

The user defines a source box around the area he wants to delete. Here continuous drawing of the source box would be very useful. If it is a valid graphical box it is drawn with a dashed line. All FNs, CASEs Brackets, indexes and text boxes which are totally (ie including pins) within the box and any outer pins on its boundary are deleted, together with all segments of wires attached to any of these objects.

The rules governing how the wires are deleted are almost the same as those for deleting them explicitly. The whole wire will go if the output to node segment is within the area otherwise all fanned out segments "downstream" of any nodes in the area will be deleted. Only whole legal (insertable) wires are actually added to the stack.

This means that the delete process for wires has two results. The first is the removal of all wiresegs outside the box which are attached to or "downstream" of those inside so that valid wires are left outside the box even if some of their wiresegs have been deleted. The second is inclusion in the group added to the stack of all the deleted wiresegs which form valid wires of any complexity and which lie entirely within the box. Deleted wiresegs which are not included in the valid wires in the group are lost from the data-structure. See also Section 6.7, Insert.

Indexes inside the source box will only be added to the stack if they are still connected to a wireseg on an insertable wire. Indexes outside the source box but attached to wires within it will be deleted if both wires go but left on the second wire as usual if one is deleted. If the text of an index extends outside the box but both its start and end pins are inside (only possible for a vertical index) the index will be deleted.

Names outside an object, eg pin names on a bracket are picked up even if part of their representation is outside the source box because they are attributes of the object.

Once the temporary box is drawn the user is given the choice to continue or quit. The group of objects is added to the stack of remembered elements as a single unit.

6.6) Duplicate Object
6.6.1) Duplicating individual objects
Static menu "DUPLICATE Object"
This is a context sensitive command.

Duplicatable objects:
These are FN instance, CASE clause, bracket and text box. All are added to the stack. The cursor on a pin picks up the object the pin is on.

Instance names, outer pin names and names on outputs of instances (LETs) have a number added to the name on both the original and the copy at the time of duplication as it must be assumed that the user is going to insert the copy and all these names must be unique in their scope - see Sections 5.9.6 and 6.6.2 on automatic naming. Adding ".1"

to an original name may mean that it is no longer displayable as text. The system automatically contracts the name if necessary using the same rules as for making a new name so the "appearance" of the diagram may change.

6.6.2) Duplicating groups of objects

Static Menu "DUPLICATE Area"

The user defines a source box around the area he wants to duplicate. As in delete_area, continuous drawing of the source box would be very useful. If it is a valid graphical box it is drawn with a dashed line. All FNs, CASEs, Brackets and text boxes which are totally (ie including pins) within the box and any outer pins on its boundary are duplicated, together with all wires connected to those objects as long as the entire wire lies within the box.

Wiresegs between other objects which happen to cross the box, which leave the box then re-enter it and between an object in the box and an object outside are not duplicated. As for delete_area, the intention is to add to the group on the stack only valid wires where all their wiresegs lie entirely within the box. This is an easier situation to visualise than for delete_area because the original wires on the screen, inside or outside the box, are not deleted. The final set of complete wires added to the group on the stack is the same whether the user was deleting or duplicating an area.

Indexes are duplicated if they are completely within the box and are connected to at least one wireseg on a wire which is to be added to the group in the stack.

Names outside an object, eg pin names on a bracket, are picked up even if part of their representation is outside the source box because they are attributes of the object.

It is unrealistic to insist on names for certain components then provide no automatic naming structure when those components are made by duplication. The naming structure specified here is intended to be as simple as possible to avoid intensive searching of the name space after each duplication.

Instance names, outer pin names and names on outputs of instances (LETs) have a number added to the name on both the originals and the copies at the time of duplication as it must be assumed that the user is going to insert the copies and all these names must be unique in their scope.

Each element of the duplicated set and the original component has the original name as a prefix to its new name. Each element is then suffixed by a point and a number which shows which set it belongs to. For example, duplicating a FN whose instance name is fred will lead

to two FNs called fred.1 (the original) and fred.2 (the copy). Duplicating this group when fred.2 has been inserted would then give fred.1.1, fred.2.1 for the original group and fred.1.2 and fred.2.2 for the copy. All names are still unique and, when a new name is added by the user it only needs to be checked against the prefix as far as the first ".". Note that "." is a forbidden character in real ELLA names. We have chosen it here because it won't impose restrictions on other user defined ELLA names. The obvious legal choice would have been "_" (visible space). However not allowing "_" in any name, particularly names like fred_1 in case a FN called fred was ever duplicated, is far too restrictive. When we translate the graphics to text all "."s are replaced by "__" (two visible spaces). This is a valid ELLA string which will, of course, have to be forbidden in user-defined names but we don't expect this to be a problem, in fact no-one may ever notice it! If they do we can continue to add "_"s or another unlikely combination of legal characters (such as "_*"?), to our duplicated names until the objections stop.

The original names which are now extended by ".1" may no longer be displayable as text and will be contracted automatically by the system.

Once the temporary box is drawn the user is given the choice to continue or quit. The group of objects is added to the stack of remembered elements as a single unit.

6.6.3) Duplicate object or area n times

```
Static Menu "DUPLICATE object | n "
            "DUPLICATE area | n "      { Cursor on the "n" }
```

These commands open a window for the user to type an integer. They are equivalent to the same number of individual duplicate or duplicate_area commands, except the names on the duplicated objects added to the stack have .3, .4, ..., .n+1 as suffixes.

For example an instance called "mux" duplicated three times using the single duplicate command would have "mux.1.1.1" as its original name and "mux.2", "mux.1.2" and "mux.1.1.2" as the copies. They are still obviously copies of the same instance but the names have got a bit out of hand. Using duplicate_n with n=3 would give "mux.1" for the original as before but the copies would now be "mux.2", "mux.3", "mux.4". The same rules apply for each name extended by the duplicate_area_n command.

Each element of the set of n duplications is added to the stack individually so they can be inserted individually later.

6.7) Insert Object
Static Menu "INSERT OBJECT".

Objects in the stack of deleted or duplicated elements can be re-inserted into the picture in different positions. The stack can hold either individual FNs, CASEs, Brackets and Text Boxes or groups of objects, including outer pins, wires and indexes, put there by delete_area or duplicate_area. Names in all the objects put in the stack by a duplicate procedure, either individually or in a group, have already been transformed automatically to ensure that they are unique in their scope.

6.7.1) Inserting individual objects
Individual objects are put back by clicking on a grid point which becomes the top-left corner of the box of that object, or its top or left point in the case of brackets. As usual when adding an object to the diagram it cannot overlap any existing objects. Names extending outside the inserted objects, (instance and pin names on brackets), are regenerated as if they were new so they may be displayed as text or as a symbol depending on the space available in the new position.

Brackets whose pins are named or pin-indexed may not be insertable in a few places where the names or pin-indexes overlap with nearby objects. As names cannot be implicitly lost or invisible the insert will not be allowed. The user is given a message and must try again somewhere else or move the object in the way. Temporary brackets are not made here - it is up to the user to put the bracket back in a legal place.

6.7.2) Inserting groups of objects
For a group of objects the cursor position represents the top-left of the displaced source box and inserted objects have the same position relative to the enclosing box as they did originally. The destination area of the whole box, including its boundary, must be completely clear, including clear of names. When the user clicks on the top-left grid point of a proposed destination box and the area within it is not clear, it would be helpful to draw the box and put up a message "This area is not clear, press result to continue or select to quit" so that he can study the area and decide whether to quit and empty it or result to try the box somewhere else. This is particularly true if only names would need contracting to clear the area.

In many cases, such as duplicating blocks to build a systolic array from scratch, the destination area will obviously be clear and the top-left of the destination box will then be sufficient to position the new components. It is therefore not necessary to draw the destination box and ask to continue before every insert.

Wires in a group all lie within the box - this is arranged when the group is added to the stack in the duplicate_area and delete_area procedures. (Go_back can be used to recover any deleted wires or wiresegs which have been deleted but are not part of the valid wires in the group, though, of course, the entire deleted group returns - not just the wires.)

Indexes in the group will always be reinserted as they would not have been added to the stack if they were not attached to an insertable wire. (Because of this restriction, however, all the indexes in the original area may not have made it to the stack - go back must be used to recover them if they have been deleted.) The text in the index parts is regenerated as new and may, therefore, extend outside the destination box if there is room.

Outer pins which are picked up in a source box may only be inserted on the outer function boundary. If the destination box does not touch the outer function in the right place for any outer pins in the group, they and all wire segments connected to them (plus others which would now be meaningless as in delete_area) must be deleted from the group before reinsertion. A message here with a visible destination box may be helpful as above to give the user a chance to study the area.

When the object(s) are inserted their names should be regenerated using the original algorithm for making names so they may extend outside the destination box if there is room for the full text display.

There are a few cases where names or pin-indexes on brackets, outer pins and indexes (which have to live outside their object) stop that object being inserted in certain positions. For example, a bracket with named or pin-indexed minor pins cannot be inserted on the grid line next to a named outer pin or a named index pin if the names clash. Names must not be lost and cannot be made invisible.

If an outer pin cannot be inserted this will be because it already has a contracted name or pin-index (or the name/pin-index was in full in the diagram, overlapped with something there and the system has already contracted it) and the circuit name is in the way. In this case, the circuit name is contracted and the outer pin can now be inserted.

Brackets and indexes which cannot be inserted even though the box area is clear will lie next to the border of the enclosing box which is itself close to another object. In these cases the bracket or the index is drawn as a temporary structure where an empty dashed box encloses the pins of the index (though not necessarily the text of a vertical index) or a dashed line replaces the bracket line and its pins are omitted. Wires are drawn to these objects as usual. A message appears pointing out the presence of temporary objects to the user,

who must move them - using the usual move procedures - to a valid place where they are drawn, with names, in the proper way.

6.8) Move Object

Static Menu "MOVE OBJECT".

Moveable Objects:
FN instance
CASE
Bracket
Index
Text Box
Pin
Node

The move procedure is context sensitive. After clicking on Move Object the user clicks on a moveable object which is moved repeatedly until the user clicks result. He is then offered the choice of exiting if the move is OK, or quitting if it is not by clicking on the select or result buttons on the mouse. (The prompt appears in the display area.) This gets him back to move_object and he can then go on to select another object to move. A final press of result in move_object, after resulting from a set of moves on one object, will get him back to the editor.

Objects are selected in the usual way by clicking on part of their visual representation. Where there is ambiguity, eg on brackets, a menu appears as usual. The next selected grid point becomes the top-left of the new box, or the top or left of a bracket, the new input position of an index or the new pin position on the same object.

The original position of all moving objects, except pins and nodes, is marked by a temporary (dashed) box which may be completely or partially visible after the move, depending on the distance moved. The temporary box is either the same as the object's box, or a box enclosing the pins for indexes and brackets. The original position of a pin is marked by a spike as for a minor pin on a bracket, but double thickness so there is no confusion on brackets themselves. The original position of a node is marked by a marker box centered on the original grid point.

When an object is connected to others by wires, each move calculates the new positions of the wiresegs from their original positions, not from intermediate ones.

The algorithm for moving wiresegs is a very simple one which acts only on the actual wireseg connected to the pin and, indeed, only on the straight line section of that wireseg touching the pin itself or

the next orthogonal one or both. Where there is more than one line in the wireseg the first two sections' lines can lengthen or shorten according to the movement of the pin - the minimum movement being done each time. A wireseg with a single straight line section connecting the moving pin with another pin or a node is split at its mid point if the pin moves orthogonally to the wire. No attempt is made to reroute the new wire if it overlaps any objects - instead it is drawn as a dashed line, one or two pixels wide like the original, and is called "temporary". All temporary wiresegs must be rerouted legally using a reroute_wireseg procedure (Section 6.18.7) before the picture represents valid ELLA and can be passed to the compiler.

Pins can only be moved on their own object. A minor pin which has been placed automatically on a bracket is moveable but the user needs to be warned that pin-indexes may now be required and, if the move is kept, the bracket may need redrawing with a single thickness line instead of a double one. (Section 5.6.) Pins are not added to the stack so there could be a problem where all the pin positions are used but the user wants to shuffle them. Expand box/bracket can be used to get extra pin positions until the shuffle is complete, then contract box/bracket to return it to its original size. See section 6.9. It is possible that the object itself may also have to be moved to allow the expansion. As a last resort the whole picture could be expanded and contracted again when rearranged.

FNs, CASEs, Brackets, Text Boxes and Indexes can be moved anywhere their box or line will fit in the picture. Wires attached to the pins are moved using the same algorithm as above on the wireseg touching each pin. Indexes are moved by selecting any point between and including its pins. Indexes are not rotated by moving.

Move_object cannot be called on an index pin - these are moved by the expand and contract procedures for indexes, or with the whole index.

CASE pins, including test pins, can be moved - with the usual restrictions on where they can be placed after moving.

A node can be moved anywhere along the wiresegs which join to it. This may result in temporary wiresegs which can be rerouted as usual.

Instance names, pin names and pin-indexes on brackets, pin names on indexes and pin names and pin-indexes on the outer function will be displayed in the new position if there is room, otherwise they are automatically contracted. The box of an index always surrounds its pins but, for a vertical index, all the text may not lie inside the box. When the new position of the index is chosen the text may have to be contracted to fit, exactly as for a new index. Names within an object are displayed as before.

If there is no room even for "*" or "[]" next to the pin (only possible with brackets, indexes and outer pins), the move will fail for outer pins - with a warning to try again or quit - or temporary brackets or indexes will be drawn with a warning to try again, quit or result to move the object in the way.

6.9) Change Size of Object

Static Menu "CHANGE OBJECT".
Objects whose graphical size can be changed are FN, CASE, Text Box and Bracket.

The user should click on the static menu, then click on the object on the corner of its box, or the end of its line, which is to move. If the cursor is not on the corner or end of a valid object, the system prompts the user and waits for the next try or result. Like move_object, the system keeps changing the same corner of an object until result is pressed, when he is asked to select a new point, on the same or a different object, to change. A further result gets him out of change_object.

If an object has been chosen, the user clicks on a grid point for the new position of that corner. The corner must keep the same relationship with the rest of the box or line, ie bottom right must stay bottom right etc. The new box or line is subject to all the graphical and ELLA constraints of an original box or line, eg it must not be too small, must not overlap existing objects etc. Pins will not move other than horizontally or vertically to lie on the new edge position of the box therefore a box or line cannot be contracted if any pins are outside or on the corner of the proposed new box or line - they must be moved explicitly first. A bracket cannot shorten from one end to past the major pin position without moving the major pin first - the major pin position defines right/left or top/bottom of a bracket.

Names and pin-indexes will be repositioned as if the object were new. This may lead to a different set of hidden and full texts as the new size of the object and the order of creation of the pins (not the order in which they were first named) affect the visibility of the names. This is unlikely to matter - a new object size will mean the user could easily want to redisplay the texts.

6.10) Other Operations on objects and attributes of objects

6.10.1) Show Specification

Works on instances (from the library or local) and on other

signal-changing objects in the diagram. Thus show_spec on a FN instance displays the spec as in the library, on the outer function displays the spec the user has built up in this circuit diagram, on an index displays the input and output pin TYPEs and on a bracket displays the major pin TYPE and the collateral of minor pin TYPEs. TYPEs or names which are not yet known are shown by "?" or another symbol not allowed in ELLA text. Thus possible displays may be:
FN AND = (bool: a b) -> bool.
FN CCT_NAME = (bool: a, bool: ?, ?: c) -> (int, ?).
INDEX = (?) -> (bool, int).
BRACKET = (word) -> bool, bool, bool.
BRACKET = bool, bool, bool -> word.

Text box inputs are supplied by names used in the text. The text has already been compiled so a specification, perhaps in the form of a dummy function spec or perhaps just a list of input names and an output TYPE, should already be available. Thus show_spec would deliver, depending on the chosen implementation of Text Boxes:
TEXT BOX = name1, name2, name3 -> (bool, int).
or
TEXT BOX = (bool: name1, int: name2, bool: name3) -> (bool, int).

Note that these are not exactly equivalent to ELLA FN specifications but are meant to give information relevant to each graphical object. Thus the three bools in the first bracket's input are deliberately not bracketted, as that would imply an ELLA structure and the graphical bracket has explicitly removed the structure.

6.10.2) Show formal parameters

Works on instances and makes the formal parameters from the declaration into input pin names of the instance, where the input pin doesn't already have a name. Warning message if all input pins already named as procedure will then appear to do nothing.

6.10.3) Show CASE

Works on CASE clauses. Attempts to show CASE clause so far as ELLA text. "?"s appear where types or inputs are not known.

A test whose sort is not set (still shown as #) should be added as ?:?, or ?: if the test has a pin, in the OF branch (a convention), rather than left out altogether.

Where inputs are joined the text will be a let name, or of the form "instance_name[output pin-index]", or ? if the text cannot yet be determined. This will be the same code as appears in the translator to text eventually - though that will only be given completed circuits.

6.18.4) Split pin

An output pin of known TYPE on any object except an index and a bracket should be splittable into its component TYPEs - one level of structure at a time. Brackets are not included because the output is either the major pin, which is indivisible, or a minor pin which must be one level of bracketting down from the TYPE on the major pin so is not splittable without another bracket.

A chooser pin (input) on a CASE clause can also be split into its component fields. It is not possible to add extra inputs using make_input - the compound input must be deleted and replaced by a new one including the extra fields as part of the split_pin procedure.

Selecting a splittable pin should generate a menu showing each internal TYPE in the same way as a FN spec shows the pin TYPEs. These are placed on the same box by the user, with checks done to ensure that all the fields are used once.

Pin-indexes, eg [3], will be necessary if two components have the same TYPE, or perhaps mandatory anyway - these can be placed automatically.

Splitting can only be done on an unjoined, unnamed pin of specified TYPE. A field of a split pin cannot be deleted. The user must be given a message to delete all fields together or quit.

6.18.5) Place minor pins

A bracket whose major pin TYPE is known can have its minor pins positioned automatically using the convention that the pin-indexes of the minor pins with respect to the major pin will increase from top to bottom or left to right along the bracket line and all the minor pins will be on the opposite side to the major pin.

The algorithm will centre the pins on the bracket line and change_object will be called to shorten the bracket from either or both sides if there are more grid points than pins. If the major pin is way off centre, change_object will not let the bracket shorten beyond it, regardless of the position of the minor pins.

An attempt to lengthen the bracket will be made if there are not enough grid points. The system tries to lengthen the bracket putting half the extra length on each end, then it tries lengthening from the bottom or right end, then from the top or left end. If all these attempts fail the automatic placement of the pins fails. If one is successful the pins are drawn on as spikes in the usual way, but no pin-indexes are necessary.

A double thickness bracket line should be drawn to show that the convention is in force.

If the user tries to move any of these pins, the system must spot that pin-indexes may now be required and it should redraw the bracket as a single thickness line with the necessary pin-indexes if the move is completed.

6.10.6) Formal to name
This works on the input pin of an instance and makes its formal parameter into its pin name. It can be called on such pins repeatedly until result is pressed. The formal text is now identical to a user defined pin name and the rules for placing it as text or * are also identical.

6.10.7) Reroute Wireseg
This is used to reroute temporary or permanent wiresegs along legal paths. Temporary wiresegs cannot be made by this procedure. The user selects any point on a wireseg except a fanout node which is ambiguous - a warning message tells him this and asks for another point. A dangling wire cannot be rerouted - it must be completed first.

When the wireseg has been selected the system picks which end to start from (output - input on Outer FN - or fanout node) and marks it with a small square. The user is then asked to indicate the new route starting from the marked point and ending on the other end of that wireseg. The route of the new wireseg is marked in the same way as drawing a new wire. When the route is complete it is checked for validity in the usual way and, if it is OK, the old route is deleted and the new one drawn. If the new route is illegal a message is given, but nothing is done, and the user can reselect the same wireseg for another try, or select a different one. As usual he remains in the procedure until he presses result.

6.10.8) Show direction
Works on wiresegs. The user clicks on a wireseg and holds down the select key. Bold arrows appear on the central grid point of all line sections of that wireseg until he releases the key.

6.10.9) Show import context
This works on imported instances. After selecting the operation the user clicks on the bottom left corner of an imported instance, where there is a small I symbol. The system displays the name of the context the user has imported it from.

An import whose context has not yet been set has a small "?" in its bottom left corner. Clicking on that causes an error message.

6.10.10) Change input order

This works on the outer function, on Imports made as empty boxes and on local functions (but not in the prototype), and is used to get the specification of the new function into the form the user requires. By default the input pins will appear in the specification in the order in which they were made. When change_ip_order is called the user will be shown a menu of the existing input pins, in order of creation, in a form as near ELLA as possible (see Section 6.10.1). The menu should not obscure the object the pins belong to. He will then be asked to select in turn, the first pin, the second pin, and so on up to the last pin, by moving the cursor to each line of the menu, Each pin must be selected in this way once and only once. When an input pin has been selected its line in the menu is displayed in reverse video and cannot be reselected. (It is also possible that the user would like each line in the menu numbered in the new order once he has selected it. This could be done automatically.)

Alternatively, it may be better to allow the user to enter numbers on each line of the menu in any order, rather than deducing the numbers from the selection order. The decision between these two methods can only be made after trials with many-pinned FNs.

The order of the output pins is set by pin-indexing which is compulsory where there is more than one output.

6.11) Operations on Pictures

These work from the static menu without the user having to move the cursor into the picture area. Parameters, if required, are written in a window generated alongside the command rectangle in the static menu. Error messages are written to the usual display line and text output to a temporary window which is visible until result is pressed (eg the information in show_context).

6.11.1) Search

A search facility must be provided to pick out instances of certain FNs or names by text, or other ELLA objects such as CASEs, Brackets and Indexes. This should be combined with a facility which will find, say, a CASE clause with pin name "chooser", or a bracket with local instance name "**bkt_1** ". Search will put the cursor on each object in turn if more than one instance of the class is found. The user should be able to press a "tab" key to step through these at his own speed.

If there is no instance of the class on the screen at the time the search procedure is called, it must also scroll the picture so that the first instance (nearest the origin) is visible immediately. As the user steps through the list if there is one, ordered by the distance of the top-left point of each object from the origin, the scrolling is again done automatically.

A requirement for this sort of procedure is obvious though more experience in using the system for designs is needed before a full specification is sensible. For example, there is an instance on the screen but it is not the first one found. How does the user get to see all instances? In text files a search procedure either operates on the closed file and generates a start-to-end list of all instances, or operates on an open (visible) file finding all instances from the cursor position to the end. In graphics we have a 2-dimensional search which is less intuitive. A possible solution to the open picture (cf open file) case would be to highlight all instances on a reduced size version of the whole picture while leaving the working picture, with the cursor on the first visible instance if any, in the same place. Where would this smaller picture be displayed? Or scroll the picture so the first instance is visible and tab through the list without an overview.....

6.11.2) *Highlight*

Some means of highlighting items the user has asked to see must be implemented. For example, "highlight all instances of DELAY1" should flash the graphical representation of DELAY1 on and off, put the box enclosing it in reverse video or some similar (bearable) change in display which makes all the instances obvious. This procedure is similar to search but all the instances in the class are made visible at once. If some of the class are off the screen a warning should be given so the user can scroll around and see them all. A reduced size overview may also be useful here.

6.11.3) Change Picture Size

This must be available to allow the user to make his entire picture larger if he runs out of space or smaller if he finishes a design and wants to tidy it into a compact rectangle for storage or display. The same sort of rules should apply as to changing the size of boxes within the picture - not contracting over pins or other objects, keeping the moving corner the same in relation to the new picture etc.

The picture cannot be contracted beyond its minimum size of 6x5, or 5x6 grid squares.

6.11.4) Magnification

We are not specifying zooming in this system, because of the well known problems of interactive scaling, particularly of fonts and bit patterns. Instead we are allowing two user-defined scaling factors at any one time, one for enlargement and one for reduction of the picture. The bit patterns and lines can then be scaled once when the factors are given, but we are still avoiding font scaling by specifying three set fonts. (See Sections 2.1 and 3.4.) The spacing of the objects in the picture and the sizes of the fonts available at each magnification have been designed to allow all names, or symbols, to be clearly visible.

At the original size (magnification 1) there are two fonts for text in the prototype. One has a character size of 9x13 for instance and FN names and the other is 7x9 for pin names, pin-indexes and index text. (The resolution of our screen is 100 pixels/inch - I give the character sizes so similarly sized fonts on different screens can be used.) The enlargement should use the 9x13 font for its pin names etc and a larger font again for the other names. The reduced picture should use the 7x9 font for its instance names etc and other symbols (see below) for pin names etc.

The user clicks on the "magnify" section of the static menu then enters the magnification factor as a parameter which may be in the form of an integer, 1/integer or a new grid spacing larger or smaller than the original spacing - this is unspecified as it is unclear yet which form will be most acceptable. The original grid spacing depends largely on the host machine's pixel density such that each grid point is easily pointed to by the mouse and the bit patterns defining objects such as pins are easily identified. The scaling factors, or new grid sizes, should be integral multiples or factors of the original spacing. This limits the possible scale factors for the reduced picture so, perhaps, the factors of the original grid should be given as the only choices. A grid of 4 pixels is the smallest spacing possible without indication of the presence of any information being lost.

If the factor is greater than 1 the user is also required to decide whether the graphics window size remains the same or is changed to show a different proportion of the picture. (Changing the window size also involves scaling the static menu, of course.) A magnification of less than one is used for an overview which appears with the existing display.

The system should then perform all the scaling calculations, "link" in the two relevant fonts and redisplay the new data-structure accordingly.

The magnification factor greater than one is available to the user for editing at a larger scale and for displaying all text in full. The factor chosen may still not allow all names to be displayed or a

resource dependent maximum picture size may have been reached before all the names are visible. The user can ask if there are any hidden names and is then prompted to rename or abbreviate any such names in order that a fully annotated picture is always possible.

The magnification factor less than one, given as 1%integer or as a new grid spacing less than the standard grid, is provided to allow an overview of the diagram. This can be visible at the same time as one of the other magnifications to provide a view of the whole circuit and, when visible, may be used by certain operations - see, for example, Sections 6.11.1 and 6.11.2. This picture cannot be edited itself but, if it is visible, should be updated each time the user returns to the static menu after completing an operation. The cursor position and, possibly, the visible area in the editable picture should be marked on the overview - again updated after each operation or after scrolling.

In the unlikely event that the original picture is so large that the minimum grid size for the reduction still does not allow it all to be visible at once, the overview should be scrollable in its window, otherwise the whole picture is visible and scrolling is not possible.

In the overview the 7x9 font now becomes the larger font for instance and FN names and all text in the smaller font is represented by a single smaller character or bit pattern, eg a very small "▪", "x", or "." which fits in the reduced size grid. The minimum size of the grid is 4 pixels (or larger if 4 pixels are not resolvable by the eye on a given screen). In this case all text is shown by a dot in the appropriate place for hidden text of each sort, so that no information about the presence of text is ever lost. The presence of unjoined pins can also be marked by a dot - this is the theoretical minimum size, and is not meant to be the suggested practical minimum!

The overview should be displayed at the bottom left of the graphics window on top of the picture area with options provided to put it underneath the picture area, close the overview window to an icon in the control area, or remove it altogether. (A particular windowing package on the host machine may allow the overview window to be totally or partially outside the graphics window in which case problems of overlap are avoided. This needs to be tested for a solution acceptable to a user after implementation of the overview window itself.)

The user can reset the magnification factors at any time, though there may only be one enlargement and one reduction factor at a time.

6.11.5) Go Back

This is specified to take the picture back to before the last editing command which changed its visible representation. The number of "go

backs" which should be possible from any one time depends on the storage available on the host machine and the data-structure chosen for the implementation and is therefore not specifiable here.

Because each operation repeats until result is pressed, "go back" may go a long way back.

6.11.6) Show context

This command brings up a scrollable window showing the declarations available in the context selected by the user in a format which shows their full declarations as ELLA text. Eg:

```
                CONTEXT GATES
INT n = 16.
TYPE bool = NEW(t|f|x).
TYPE nbool = [n]bool.
FN AND = ([2]bool: ip) -> bool:
FN OR = ([2]bool: ip) -> bool:
FN NOT = (bool: ip) -> bool:
MAC N_AND = ([n]bool: ip) -> bool:

Imports of Gates
FN AND = ([2]bool: ip) -> bool: RENAMED AND2
          from Context Ella
```

6.11.7) Save

This allows the user to save the current data-structure of the picture before he carries out a risky operation. It should be usable between any two operations initiated from the static menu.

6.11.8) Picture Text

Once a picture has been compiled it is stored in the ELLA system as an Intermediate Language representation of its "text" and a set of attributes representing the graphical information. The user can ask for a text file to be generated from the Intermediate Language representation using a currently available text generating program (D.J. Snell). This file exists independently in the operating system - not in the ELLA system.

6.11.9) Saving a Compiled Graphical Declaration

If a new declaration is to overwrite an existing graphical declaration the user should have the capability to save the existing declaration in a file generated from the Intermediate Language representation of the declaration and its graphical attributes. The Picture Text operation above would form the first part of this file, followed by some reconstructable representation of the attributes. This will

protect the user from losing a graphical declaration completely in the event that he has already lost the original picture file. A procedure to regenerate the graphical declaration from the new file must also exist.

6.11.10) Give cursor coordinates
This displays the current cursor coordinates on the display/warning line - probably in units of the grid spacing. On the prototype we have a grid spacing of 10 pixels and a pixel size of about 100/inch. These numbers mean it is easy to relate pixel coordinates to position in the picture. With other grid spacings and screen resolutions this may not be so intuitive so grid units should be used.

6.11.11) Move cursor
This takes an (x,y) coordinate pair typed in a text window as a parameter and moves the cursor to that position, including scrolling the screen so the cursor is centralised if the position is not already visible. The x and y integers should be in grid units - see Section 6.11.10.

6.12) TYPEs
6.12.1) The TYPE Checker
TYPE checking should be done interactively so that no inconsistencies are ever possible. In addition the data-structure must know whether all TYPEs have been filled in before passing the picture to the compiler. This can be done within a "graphical compiler", which pre-processes the picture to check that it is valid ELLA, before the picture is entered in the ELLA data-base.

All the rules followed by the TYPE checker are identical to the rules for text ELLA so there should be no surprises for the user. This includes TYPE equivalence such as TYPE char = [8]bool and TYPE word = [8]bool implies char=word=[8]bool for joining purposes. Also TYPE fred = bool is supported. Thus a pin must know if its TYPE is fred or bool as the names are (presumably) added to aid understandability. Within the picture there may be structures of TYPEs which are not named in the library - this causes no problems since all the basic TYPEs have been declared in the library.

The TYPE is a property of each pin. Pins on instances know their TYPE when the instance is made and these TYPEs cannot be changed. Pins on other objects have TYPEs set by the user, either explicitly or implicitly as he builds up the picture. Whether or not the TYPE of a pin is specified is obvious from the graphical representation of each pin.

When a new wire segment is made the TYPE at each end of the wire segment is checked and must match.

If a TYPE on one end of a proposed wire is not set, that pin is automatically given the TYPE of the other end. At that point, unless the unset pin is on the outer function where the user can build up whatever specification he wants, a check must be done on other pins on the same object to ensure that the new TYPE is compatible.

On brackets the TYPEs on the minor pins must "add up" to the TYPE on the major pin which is, of course, always a structure. Whichever TYPEs are known first, it is possible to check that the new addition is compatible.

On an index, the TYPE on the input pin must be a structure and the output pin can be a structure or a simple TYPE depending on the input and the index text. An index cannot exist without at least one of its pins joined, so the text can always be checked against one or both TYPEs when it is written. This will be a complete check if both TYPEs or just the input TYPE is known, but only a partial check if just the output TYPE is known. If the text is written before the second join, the TYPE on the new wire must be compatible with the TYPE deduced from the other pin and the text. The index is flagged as incomplete until both pins are joined to other pins of known TYPE and all the joins are consistent.

On a CASE clause the TYPEs on the chooser, output, test pins and test texts are interlinked. The TYPE of the basic values in the test texts must always be "chooser TYPE: output TYPE", "chooser TYPE:", "output TYPE" or "" depending on the presence of a pin and the sort of test. Thus, writing a test may set the chooser and/or output TYPEs and, conversely, setting the chooser TYPE or the output TYPE specifies the TYPE of the first or second part of each test. The test pins have the same TYPE as the output or second part of a test text, so setting any one of these sets them all. Any inconsistency is flagged immediately, with a warning message telling the user what TYPEs are expected, and the user can then edit the new TYPE or test text, or change other TYPEs or tests on the CASE clause until consistency is regained. See Section 6.3.3 on adding text to CASE tests.

The output TYPE of a text box must match the TYPE of the unit delivered by the text in the box - this is enforced interactively.

If both TYPEs at each end of a proposed wire are known and do not match, or if one is known then the other is deduced and found to be incompatible with other TYPEs on the same object, the set of points defining the line of the wire is lost and the user must start again

with a new wire or result to do something else. It is not possible to join between two pins whose TYPEs are not known.

Specifying the TYPE of a pin explicitly by writing the name (or number) of the TYPE in a window is subject to the same consistency checks against other pins on the same object, or against text within the object if appropriate.

If a TYPE is set implicitly, eg by a join, it remains set even if the original operation is negated, eg the wire is deleted.

Brackets used to denote REFORM are subject to different checks, equivalent to text ELLA. The input and output are both structures with their basic values in the same order but bracketed differently. (Other brackets simply add or remove a single level of bracketing.)

The TYPE checker must spot where ambiguity arises, eg two bool outputs from a bracket whose major pin is of TYPE [2]bool need pin-indexing to show which bool is which. The user must be told immediately when this problem is identified and the picture cannot be compiled until the pin-indexes have been added.

Where the TYPE checker sets TYPEs implicitly it is also reponsible for redrawing the pins with their specified representations.

The TYPE checker should be called whenever the user joins pins, specifies TYPEs, splits pins, adds index text, adds test text, adds text to a Text Box, adds a pin-index to a pin or makes pins on an object whose specification is being developed and is constrained by TYPEs already present.

6.12.2) Operations involving TYPEs

6.12.2.1) Show TYPEs

This displays all TYPEs currently recognised in the circuit, ie both those from the library and rows or structures used in constructing the circuit. It also gives each TYPE a number which can be used as input by the user instead of the TYPE name or, more usefully, instead of the row/structure text in the case of an internal TYPE.

The numbers are the order of creation of the TYPEs, so library TYPEs will appear first. A re-order procedure may turn out to be useful so the user can group his available TYPEs into any order he wants.

The display appears in a window next to the command rectangle in the static menu as this is an operation on the whole picture.

6.12.2.2) Specify TYPE

Works on pins on all objects, except instances whose TYPEs are

already set. It therefore includes index pins and test pins on CASE clauses. The user selects a pin and a window appears in which to write the name or number (see section 6.12.2.1) of the TYPE required.

If the TYPE is not in the existing TYPE list it will be passed to the TYPE checker as long as it looks like a possible row or structure. For example, "boool" will not be considered as a possible internal TYPE, but "(int, bool)" or "[3]boool" will - though they may be rejected later if their components are not known TYPEs. A suitable new TYPE is then added to the data-structure and numbered. If the checks fail the user can edit the window or quit.

An unspecify_type is also required as TYPEs may only be changed, not deleted, by further calls of specify TYPE on a pin - though giving "0" to specify_type may do.

If specify_type is successful the graphical representation of the pin is changed accordingly. (Similarly for unspecify_type.)

6.12.2.3) Show TYPE

Works on all pins and wires, though TYPEs are, strictly speaking, properties of pins. Displays name of simple TYPE, or name and components of a structure/row, plus the TYPE number on selected pin or wire in the display area outside the picture, eg "bool {1}" or "char = [8]bool {3}". Where there are two coincident pins, the display gives the TYPE on each, eg "Upper pin: bool {1}/Lower pin: char = [8]bool {3}".

Where a wire or pin carries an internal TYPE not declared in the ELLA context the system displays the collateral of its components in terms of declared TYPE names and gives its number, eg "(bool, int, char) {5}".

7) ELLA GRAPHICS SPECIFICATION - PART 4, UN-PROTOTYPED FEATURES

This Section contains preliminary specifications of features which are untested in the RSRE prototype but which extend the graphical language to cover almost all elements of ELLA syntax (except MACros).

The features covered here should be specified in enough detail to make our intentions clear, especially as they should match in character the Version 1 specifications in the preceding Sections. In some cases, eg bidirectionals, text ELLA is finalising the equivalent constructs so we have left their graphical spec to Version 2 when we hope the system will be more stable. Other features, eg local declarations, are dependent on the windowing system and recursive calls of the graphical editor or picture system. We felt these would be easier to specify correctly when the Version 1 implementation is well under way. Finally, doodles are an idea for a sort of unchecked sketch pad which the user may demand, ridicule, or demand then never use because he realised how much he relied on the checks. No further specification of this feature should be made until real user feedback on Version 1 and its limitations has been received.

7.1) Bidirectional TYPEs and wires

An ELLA FNTYPE is a bidirectional TYPE. It represents the TYPE carried by a single bidirectional wire or a bundle of uni-directional wires. Text ELLA cannot show the difference between these wire types but the graphics system can.

A bidirectional pin on most ELLA objects looks like an ordinary pin with an enclosing box. Bidirectional minor pins on a bracket are shown as two short spikes one pixel apart such that the spikes are covered when the wire is drawn. The input to an index must be a structure but its output may be a single bidirectional TYPE. This is represented by enclosing the > in three sides of a box (see diagram). All bidirectional pins can be of specified and unspecified TYPE. (Bidirectional outputs will be possible in the next release of ELLA and are therefore specified here for completeness.)

A bidirectional TYPE as part of a structured TYPE is represented as a double thickness ordinary wire as before.

When the user joins two bidirectional pins he is asked, or can choose from the static menu, whether he wants a single wire or a bundle. A bundle is drawn as two lines, each one pixel wide and one pixel apart. A single wire has the same representation plus a dashed pattern along the blank line in the middle. In the prototype we have tried various dashes - 3 pixels black and 7 white in a 10 pixel grid spacing looks best. Note this brings the number of line styles in the system up to four - solid, white, 2xdashed.

The wiretype of a bidirectional wire can be changed by a boolean operation available from the Static Menu. (Change_wiretype.) This works on any wireseg of a bidirectional wire and changes the representation of the whole wire.

A bidirectional wire can fanout as usual - the wiretype being the same over the whole wire.

Arrows can be placed on a bidirectional wire as usual.

Bidirectional wiresegs can be temporary due to a move operation. These have the two continuous lines replaced by dashes as usual. Reroute_wireseg can also be used.

Bidirectional pins can be made implicitly by joining wherever their TYPE can be deduced.

An unjoined pin of unspecified TYPE can be changed to the same sort of bidirectional pin by a boolean operation.

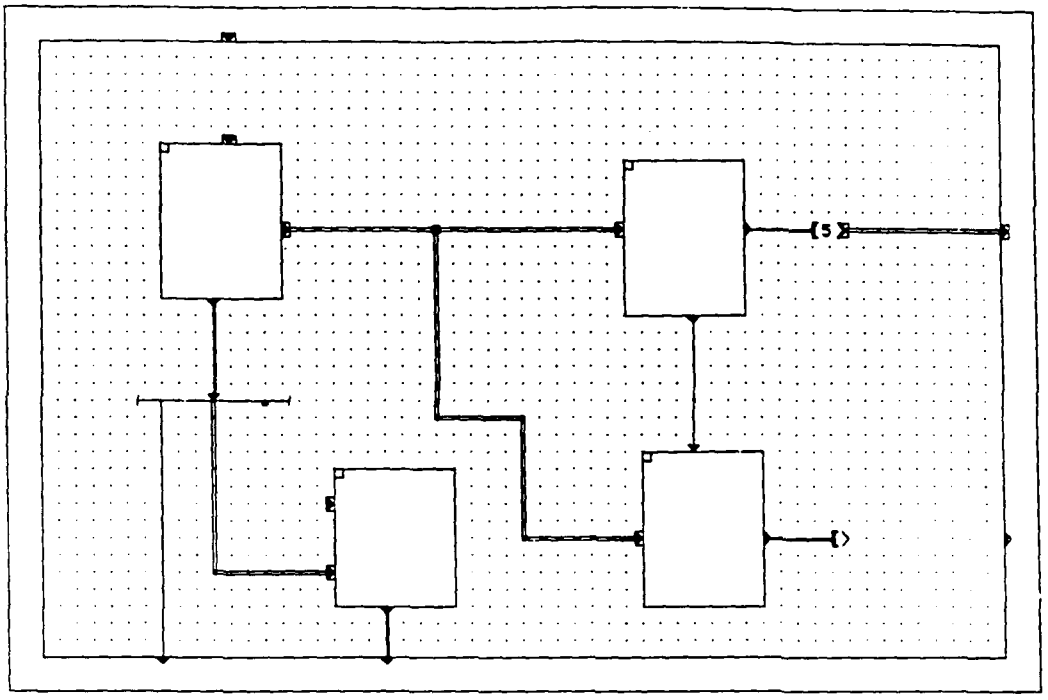
There must be some way of getting the possible signals separately from a bidirectional wire - it is likely that the two wiretypes should have different methods for this to reflect the different hardware which will be needed in reality.

A bundle of wires could be tapped using a node-like construct and the wire leaving the tap indexed (like a pin-index) to show which signal it carries.

A single bidirectional wire could be the input to a short bracket like the REFORM construct but with a single bidirectional input. The output is then the output signal of the FNTYPE.

These two constructs are not fully considered.

A graphical representation of the plug-and-socket model of FNSETs has not been thought out.


5

7.2) Associated TYPEs

No operation has been provided or specified to associate or de-associate TYPEs. A FN could, however, be declared textually and compiled into the working context which would do the required operation and could then be instanced in the picture.

We should also provide a graphical operation, for example a small box with a single input and output pin and the "&" or "//" symbol visible in the box depending on the operation. The box should be a minimum of 2x2 grid squares in size to allow for the input and output wires joining on any side. The size of the box should be changable to allow the text identifier to be visible as well as the operation symbol, eg "//data" or "data&".

To make a & or // box the user should press result on an empty grid point while he is defining a wire and the existing pop-up menu should be extended to give him the extra option "& or //" as well as "index" or "quit". This will automatically draw a box, if the area is clear, with the right symbol inside and a pin of known TYPE where the wire joined - ie at the cursor position. The TYPE at the start of the wire must be known, as for making an index, or we would end up with a join between unspecified pins. The second pin is added later on any of the three remaining sides.

If the TYPE on the first wire is already associated, the system will draw a //-box if the starting point was an output (or equivalent) or a &-box if it was an input. In these cases the TYPE of the other pin on the new box can be deduced and is set either immediately, if the pin is present, or as soon as the pin is placed. Conversely, if the TYPE on the first value is not associated, the system will draw a //-box if the starting point was an input or a &-box if it was an output. The TYPE on the second pin is set by a join or by specify_type on an explicitly made pin and must have the same basic TYPE as the first pin.

Show_spec should work on these boxes as usual, and show_type on the pins. In addition the user should be able to click on the central grid point of the box and ask for the "text" of the associated TYPE, for example "data&[8]bool". As the box with its "&" or "//" symbol will still be visible this is sufficient information.

7.3) Local Declarations

7.3.1) Local TYPE Declarations

These will allow the user to declare a TYPE inside the FN being declared, exactly as in text ELLA, and should be possible from a command in the static menu. A single character high , sideways scrollable window should appear in which the user types his text as

usual. Possibly the word "TYPE" should already be in the window with the cursor after it.

When the user presses result to close the window, the text is checked and, if it is correct, the window closes and the new TYPE is added to the usable list of TYPEs with a flag to the effect that it is locally defined (name in bold type?). It should be used exactly as in text ELLA - that is the TYPE name used after the local declaration will pick up the local definition, used before it will pick up the name from the working context.

If the text was not correct, the window remains open with the cursor at the first mistake and an error message on the warning line. Once the user has steeped through all the errors and tried to correct them he can press result again to repeat the checking process.

7.3.2) Local FN Declarations

It is intended that the user can ask to define a local FN within the picture he is already constructing. A second window should appear whose size is user defined but which should be no bigger than the existing picture area so that the same static menu can be used. (Or just a whole new picture system, menu and all, should appear.) This is a recursive call of the picture editor if the static menu is shared or, if not, a recursive call of the entire graphics system.

The user draws his local FN in the same way as the outer one and, when he closes the window, the MAKE instance procedure is called in the usual way, once the local declaration is passed as correct and entered into the data-structure of the outer declaration.

The FN_name while the local circuit is being declared, and its FN_name and instance_name when the window is closed and the local instance is drawn, are written in bold or have solid symbols. The FN_name of the local FN follows the same rules as for text ELLA. Thus once a local FN has been defined, subsequent uses of its FN_name will instance it, even if instances of an outside FN with the same name have already been made. The use of normal and bold TYPE for the names will differentiate between the two sets of instances.

When the circuit is "compiled" the presence of a local declaration is spotted and it should be checked and stored in exactly the same way as a nested textual declaration.

It should also be possible to declare a text FN within the inner window, so "text" or "picture" should be a parameter to the local FN procedure. The text is written in the usual way. DELAYs or RAMs may often be defined locally as text - though see Section 7.3.2.1 for easy graphical declarations. The system should spot a text declaration

whose body is a DELAY or RAM and insert a bold "D" or "R" accordingly - see Section 7.4.

It is assumed here that the user will not bother to define a local FN without making an instance of it immediately - he doesn't have to join it. Further instances can already be made by duplicating the original instance, and renaming to remove the automatic names if required. Otherwise another scheme needs devising for extra instances, eg point to the original instance after the MAKE command to pick up the local declaration and continue making the new instance as usual, or a set of icons for local declarations somewhere outside the picture area which the user can point to as in ELLAview.

7.3.2.1) Local DELAYs and RAMs

Certain FNs, such as DELAYs or RAMs are commonly declared as local FNs in text ELLA and help can be given in their local graphical definitions.

The user indicates from the menu that he wishes to draw a DELAY FN. A menu appears asking for the name of the FN, then the TYPE of signal to be delayed, then for the basic value(s) and integers which should be entered as for ELLA text - eg "x,1" or "x,1,f,4". The TYPE and basic values are checked for consistency and the name for scope then the FN is drawn as above. A bold "D" appears in the bottom right corner of the box. (See Section 7.4.)

Similarly a RAM FN can be defined using a menu requesting the FN name, the three TYPEs for data, addresses and enable, and the basic value of the data TYPE for initialisation. When all these are entered and checked the FN can be drawn as other local FNs above, with a bold "R" in the bottom right corner.

Depending on response from users this facility could be extended to include other "special" FNs in ELLA which are commonly defined locally (eg BIOPs?).

7.4) DELAYs, RAMs, BIOPs, Alien Code and Timescaling

Symbols in a FN box representing these types of FNs can all be placed in the bottom right corner of a box because they are all mutually exclusive.

DELAYs, RAMs and BIOPs are already in the text language. A FN whose body is a DELAY, RAM or BIOP could be identified by a small "D", "R" or "B" in the bottom right corner. The symbol can be interogated to

give the delay or RAM TYPEs and parameters as in the text declaration. (What do BIOPs look like?)

Alien Code is not yet implemented, but a FN whose body is Alien could be represented by a small "A". The text of the Alien Code should be displayable (editable?) in a window if selected via the corner.

A timescaled FN could be identified by a circle bottom right corner. The symbol is accessible via the corner and can be interogated to show the timescaling factor, eg +3 for 3 times faster or -4 for four times slower. ("T" is already used for text boxes - though in the bottom right corner.)

More discussion is needed on BIOPs, Alien Code and timescaling with the ELLA team at RSRE who are designing the features.

An instance made from the library - from the working context or imported - or locally should be identified automatically as one of these FNs and the appropriate symbol written in immediately. (What about other special FN bodies, eg CASE ("C") or SEQ ("S")? These anotations could be added if users want them.)

For more visible information it is possible to use the Abbreviation feature (Section 6.3.1.3) on the FN or instance names. For example an instance of "FN DELINT = (int) -> int: DELAY(i/0, 1)." will have a "D" in its bottom right corner and could have its instance name abbreviated to "i/0,1" if the user found this helpful.

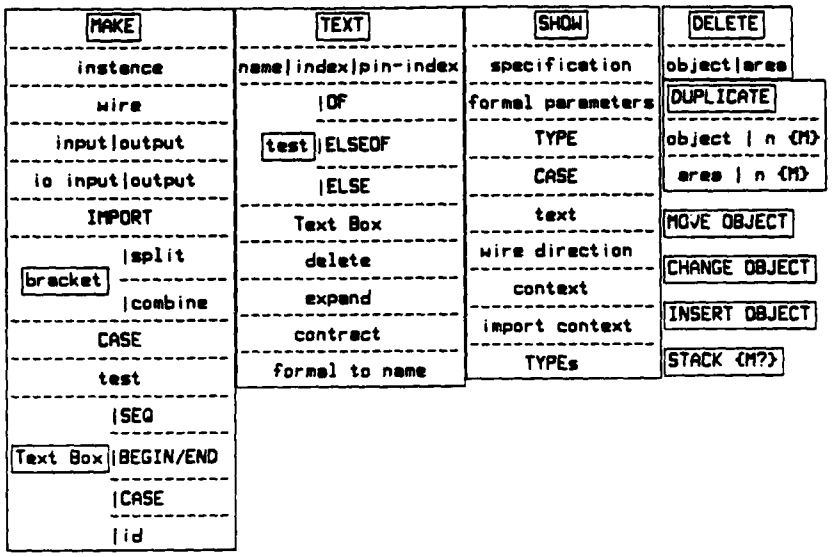
7.5) Doodles

These are intended to be unspecified (ie initially unknown) objects and may turn out to be unnecessary. Each doodle is essentially a box with nameable pins of any sort and specifiable TYPE, and a local name field. When the user thinks he knows what the doodle should be he can "convert" it to a genuine object. At this point all sorts of checks are done to see if the data-structure of the doodle will map onto that of the chosen object. For example, a Bracket must have one pin of one sort and all the others of the other sort so that the major pin can be deduced. It also needs a line representation and the pins placing on the line before the conversion is complete. Similarly a CASE clause needs its chooser identifying from the set of inputs, the remaining inputs being tests whose texts, if added, must be right. The single output pin becomes the output of the CASE, there should be no name for the doodle and the box should be redrawn as double thickness. (Alternatively, a doodle only ever converts to a FN instance and may, therefore, only be linked to a system generated local declaration or one from the working context.)

The idea of a convertible object has been around for some time and

would allow the user much freedom in setting something down on the screen as he begins each level of design. However, the TYPE checker will not have been used in the development of a doodle and this, in our strongly TYPEd and interactively checked system, may lead to too many errors needing correction during the conversion process. Whether doodles ever exist in the commercial implementation depends on user feedback from real design engineers but they do seem to be possible from our prototype Modes.

GRAPHICAL OPERATIONS



{M} denotes pull-down menu or window for parameter.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification, eg (R), (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	REPORT 89017		UNCLASSIFIED

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN WORCESTERSHIRE WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title
THE RATIONALE AND SPECIFICATION OF THE ELLA GRAPHICS SYSTEM

7a. Title in Foreign Language (in the case of Translations)

7b. Presented at (for Conference Papers): Title, Place and Date of Conference

8. Author 1: Surname, Initials	9a. Author 2	9b. Authors 3, 4 ...	10. Date	pp. ref.
WOOD C S	PEELING N E	THOMPSON J I	1989.10	53

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution Statement
UNLIMITED

Descriptors (or Keywords)

Continue on separate piece of paper

Abstract

The ELLA Graphical Language includes graphical representations of most of the high level abstractions in text ELLA, in particular high level data-types and behavioural constructs. The Graphical Editor allows a designer to draw a circuit using these abstractions, which is then compiled into the ELLA system.

Graphical and text declarations can be used in exactly equivalent ways in the extended ELLA system specified here. The Graphical input system thus gives the designer the freedom to use ELLA graphics or text where each is most appropriate, without losing the high level design capabilities when he prefers to use graphics ——>

Page — 8